Hitachi Microcomputer

Support Software

SH Series C Compiler User's Manual

HITACHI

ADE-702-095

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

Preface

This manual explains the facilities and operating procedures for the SH series C compiler (Ver. 2.0). The C compiler translates source programs written in C into relocatable object programs or assembly programs for Hitachi SH7000 series RISC microcomputers.

This manual consists of four parts and appendixes. The information contained in each part is summarized below.

(1) PART I OVERVIEW AND OPERATIONS

The overview sections cover the following:

① C compiler functions

② Developing procedures

The operation sections cover the following:

③ How to invoke the C compiler

④ Optional functions

⑤ Listings created by the C compiler

(2) PART II PROGRAMMING

This part explains the limitations of the C compiler and the special factors in object program execution which should be considered when creating a program.

(3) PART III SYSTEM INSTALLATION

This part explains the requirements when installing an object program generated by the C compiler on a system. They are the object program being written in ROM and memory allocation. In addition, specifications of the low-level interface routine must be made by the user when using standard I/O library and memory management library.

(4) PART IV ERROR MESSAGES

This part explains the error messages corresponding to compilation errors and the standard library error messages corresponding to run time errors.

This manual corresponds to operating systems that function on UNIX, MS-DOS, or IBM-PC systems. In this manual, operating systems functioning on MS-DOS or IBM-PC systems are referred to as PC systems.

Notes on Symbols: The following symbols are used in this manual.

Symbols Used in This Manual

Symbol	Explanation
< >	Indicates an item to be specified.
[ ]	Indicates an item that can be omitted.
...	Indicates that the preceding item can be repeated.
Δ	Indicates one or more blanks.
(RET)	Indicates the carriage return key (return key).
|	Indicates that one of the items must be selected.
(CNTL)	Indicates that the control key should be held down while pressing the key that follows.

UNIX is an operating system administrated by the UNIX System Laboratories (United States).
MS-DOS is an operating system administrated by the Microsoft Corporation (United States).
IBM-PC is an personal computer system administrated by IBM (United States).

Contents

Part I OVERVIEW AND OPERATIONS 1

Section 1 Overview 3

Section 2 Developing Procedures 4

Section 3 C Compiler Execution 5
3.1 How to Invoke the C Compiler 5
3.2 Naming Files 8
3.3 Compiler Options 9
3.4 Option Combinations 11
3.5 C Compiler Listings 12

Part II PROGRAMMING 19

Section 1 Limitations of the C Compiler 21

Section 2 Executing a C Program 23
2.1 Structure of Object Programs 24
2.2 Internal Data Representation 26
2.3 Linkage with Assembly Programs 31
2.3.1 External Identifier Reference 32
2.3.2 Function Call Interface 34

Section 3 Extended Specifications 43
3.1 Interrupt Functions 43
3.2 Intrinsic Functions 47

Section 4 Notes on Programming 51
4.1 Coding Notes 51
4.2 Notes on Programming Development 54

Part III SYSTEM INSTALLATION 55

Section 1 Overview 57

Section 2 Allocating Memory Areas 58
2.1 Static Area Allocation 58
2.1.1 Data to be Allocated in Static Area 58
2.1.2 Static Area Size Calculation 58
2.1.3 ROM and RAM Allocation 61
2.1.4 Initialized Data Area Allocation 61
2.1.5 Example: Memory Area Allocation and Address Specification at Program Linkage 61
2.2 Dynamic Area Allocation 63
2.2.1 Dynamic Areas 63
2.2.2 Dynamic Area Size Calculation 63
2.2.3 Rules for Allocating Dynamic Area 66

Section 3 Setting the Execution Environment 67
3.1 Vector Table Setting (VEC_TBL) 68
3.2 Initialization (_ _INIT) 69
3.3 Section Initialization (_ _INITSCT) 70

Section 4 Setting the C Library Function Execution Environment 73
4.1 Setting Vector Table (VEC_TBL) 74
4.2 Initializing Registers (_ _INIT) 75
4.3 Initializing Sections (_ _INITSCT) 76
4.4 Initializing C Library Functions (_ _INITLIB) 76
4.4.1 Creating Initialization Routine for Standard I/O Library Function (_INIT_IOLIB) 77
4.4.2 Creating Initialization Routine for Other Library Function (_INIT_OTHERLIB) 78
4.5 Closing Files (_ _CLOSEALL) 79
4.6 Creating Low-Level Interface Routines 80

Part IV ERROR MESSAGES 89

Section 1 Error Messages Output by the C Compiler 91
1.1 Error Message Format 91
1.2 C Compiler Action and Programmer Response for Each Error Level 92
1.3 List of Error Messages 93

Section 2 Error Messages Output for the C Library Functions 140

APPENDIX 143

Appendix A Language and Standard Library Function Specifications of the C Compiler 145
A.1 Language Specifications of the C Compiler 145
A.1.1 Compilation Specifications 145
A.1.2 Environmental Specifications 145
A.1.3 Identifiers 145
A.1.4 Characters 146
A.1.5 Integer 147
A.1.6 Floating-Point Numbers 148
A.1.7 Arrays and Pointers 149
A.1.8 Register 149
A.1.9 Structure, Union, Enumeration, and Bit Field Types 150
A.1.10 Modifier 150
A.1.11 Declarations 151
A.1.12 Statement 151
A.1.13 Preprocessor 152
A.2 C Library Function Specifications 153
A.2.1 stddef.h 153
A.2.2 assert.h 153
A.2.3 ctype.h 154
A.2.4 math.h 154
A.2.5 stdio.h 155
A.2.6 string.h 156
A.2.7 Not Supported Library 156
A.3 Floating-Point Number Specifications 157
A.3.1 Internal Representation of Floating-Point Numbers 157

A.3.2 float 159
A.3.3 double and long double 160
A.3.4 Floating-point Operation Specifications 162

Appendix B Parameter Allocation Example 165

Appendix C Usage of Registers and Stack Area 168

Appendix D Creating Termination Functions 169
D.1 Creating Library onexit Function 169
D.2 Creating exit Function 170
D.3 Creating abort Routine 171

Appendix E Examples of Low-Level Interface Routine 172

Appendix F ASCII Codes 177

Index 178

Figures

Part I

2-1 Relationship between the C Compiler and Other Software 4
3-1 Source Listing Output for show=noinclude and noexpansion 13
3-2 Source Listing Output for show=include and expansion 13
3-3 Object Listing 15
3-4 Statistics Information 16
3-5 command line specification 17

Part II

2-1 Allocation and Deallocation of a Stack Frame 34
2-2 Parameter Area Allocation 39
2-3 Example of Allocation to Parameter Registers 41
2-4 Return Value Setting Area Used When Return Value Is Written to Memory 42
3-1 Stack Processing by an Interrupt Function 44

Part III

2-1 Section Size Information 58
2-2 Static Area Allocation 62
2-3 Nested Function Calls and Stack Size 65
3-1 Program Configuration (No C Library Function is Used) 67
4-1 Program Configuration When C Library Function Is Used 73
4-2 FILE-Type Data 78

Part IV

1-1 Error Messages Format (UNIX Systems) 91
1-2 Error Messages Format (PC Systems) 91

Appendix

A-1 Structure for the Internal Representation of Floating-Point Numbers 157
C-1 Usage of Registers and Stack Area 168

Tables

Part I

3-1 Standard File Extensions Used by the C Compiler 8
3-2 C Compiler Options 9
3-3 Macro Names, Names, and Constants Specified by the define Option 10
3-4 Option Combinations 11
3-5 Structure and Contents of C Compiler Listings 12

Part II

1-1 Limitation of the C Compiler 21
2-1 Memory Area Types and Characteristics 24
2-2 Internal Representation of Scalar-Type Data 26
2-3 Internal Representation of Aggregate-Type Data 27
2-4 Bit Field Member Specifications 28
2-5 Rules on Changes in Registers After a Function Call 35
2-6 General Rules on Parameter Area Allocation 40
2-7 Return Value Type and Setting Location 42
3-1 Interrupt Specifications 43
3-2 Intrinsic Functions 47
4-1 Troubleshooting 54

Part III

2-1 Stack Size Calculation Example 65
4-1 Low-Level Interface Routines 80

Part IV

1-1 C Compiler Action and Programmer Response for Each Error Level 92

Appendix

A-1 Compilation Specifications 145
A-2 Environmental Specifications 145
A-3 Identifier Specifications 145
A-4 Character Specifications 146
A-5 Integer Specifications 147
A-6 Integer Types and Their Corresponding Data Range 147
A-7 Floating-Point Number Specifications 148

A-8 Limits on Floating-Point Numbers 148
A-9 Array and Pointer Specifications 149
A-10 Register Specifications 149
A-11 Specifications for Structure, Union, Enumeration, and Bit Field Types 150
A-12 Modifier Specifications 150
A-13 Declaration Specifications 151
A-14 Statement Specifications 151
A-15 Preprocessor Specifications 152
A-16 stddef.h Specifications 153
A-17 assert.h Specifications 153
A-18 ctype.h Specifications 154
A-19 Set of Characters that Returns True 154
A-20 math.h Specifications 154
A-21 stdio.h Specifications 155
A-22 Infinity and Not a Number 156
A-23 string.h Specifications 156
A-24 Libraries Not Supported by the C Compiler 156
A-25 Types of Values Represented by Floating-Point Numbers 158

PART I

OVERVIEW AND OPERATIONS

Section 1 Overview

The SH series C compiler inputs source programs written in C and outputs relocatable object programs or assembly source programs.

The C compiler supports the SH7000-series microcomputers (referred to as SH).

Section 2 Developing Procedures

Figure 2-1 shows the relationship between the C compiler package and other software for program development. The C compiler package includes the software enclosed by the dotted line.

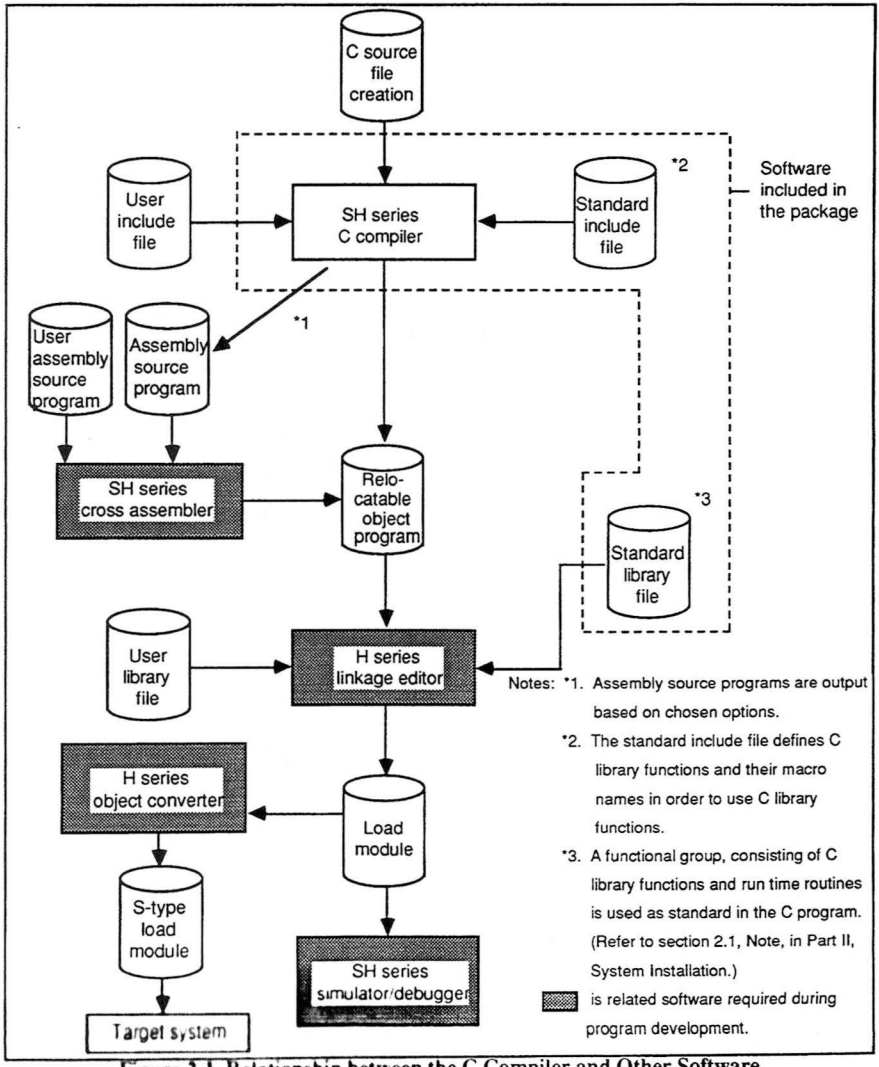


Figure 2-1 Relationship between the C Compiler and Other Software

Section 3 C Compiler Execution

This section explains how to invoke the C compiler, specify C compiler options, and interpret C compiler listings.

3.1 How to Invoke the C Compiler

The format for the command line used to invoke the C compiler is as follows.

UNIX systems:

```
shc[Δ<option>...][Δ<file name>[Δ<option>...]...]
```

PC systems:

```
shc[Δ<option>...]Δ[<file name>]
```

The general operations of the C compiler are described below.

Compiling Programs:

```
shcΔtest.c (RET)
```

The C source program test.c is compiled.

C Compiler Options (UNIX):

```
shcΔ-debugΔ-listfileΔ-show=noobject,expansionΔtest.c (RET)
```

Insert minus (-) before options (**debug**, **listfile**, and **show**). When multiple options are specified, separate them with a space (Δ). Also when multiple suboptions are specified, separate them with a comma (,).

C Compiler Options (PC):

```
shcΔ/debugΔ/listfileΔ/show=(noobject,expansion)Δtest.c
```

Insert a slash (/) before the options (**debug**, **listfile**, and **show**). When multiple options are specified, separate them with a space (Δ). Also when multiple suboptions are specified, separate them with a comma (,) and enclose them in parentheses.

Compiling Multiple Programs:

Several C source programs can be compiled by a single command on UNIX systems.

Example 1: Specifying multiple programs

```
shcΔtest1.cΔtest2.c(RET)
```

Example 2: Specifying options for all C source programs

```
shcΔ-listfileΔtest1.cΔtest2.c (RET)
```

The **listfile** option is valid for both test1.c and test2.c.

Example 3: Specifying options for particular C source programs

```
shcΔtest1.cΔtest2.cΔ-listfile (RET)
```

The **listfile** option is valid for only test2.c. Options specified for particular C source programs have priority over those specified for all C source programs.

Option List:

```
shc (RET)
```

Instead of compiling, the C compiler outputs the standard command line format and option list.

3.2 Naming Files

A standard file extension is automatically added to the name of a file when omitted. The standard file extensions used by the C compiler and related software are shown in table 3-1.

Table 3-1 Standard File Extensions Used by the C Compiler

File Extensioner	Description
c	Source program file written in C
h	Include file
lst, lis	Listing file*1
obj	Relocatable object program file
src	Assembly source program file
lib	Library file
abs	Absolute load module file
rel	Relocatable load module file
map	Linkage map listing file

Note: *1. The listing file extension is lis on UNIX systems and 1st on PC systems.

The general conventions for naming files depend on the host machine. Refer to the manual of the host machine in use.

3.3 Compiler Options

Table 3-2 shows C compiler option formats, abbreviations, and defaults. Characters underlined indicate the minimum valid abbreviation. Bold characters indicate default assumption.

Table 3-2 C Compiler Options

Item	Format	Suboption	Specification
Optimization level	optimize =	0 |	Object without optimization is output.
		1	Object with optimization is output.
Listings and formats*1	show =	**source** |nosource |	Source list yes/no
		object |noobject |	Object list yes/no
		statistics |nostatistics |	Statistics information yes/no
		include |**noinclude** |	List after include expansion yes/no
		expansion |**noexpansion** |	List after macro expansion yes/no
	*2	width = <numeric value> |	Maximum characters per line: 0, 80–132
	*2	length = <numeric value>	Maximum lines per page: 0, 40–255
		Default: **w = 132, l = 66**	
Listing file	listfile [= <listing file name>] *3		Output
	nolistfile		No output
Object file	objectfile = <object file name>		Output
Object progam format	code =	**machine code** |	Program in machine language is output.
		asmcode	Assembly source progam is output.
Debug information	debug		Output
	nodebug		No output
Macro name	define =	<macro name> = <name> |	<name> is defined as <macro name>
		<macro name> = <constant> |	<constant> is defined as <macro name>
		<macro name> *4	<macro name> is assumed to be defined.
Include file	include =	<path name> *5	Include file destination path name is specified. (Multi-specification is possible.) *4
Section name	section = *5	program = <section name> |	Program area section name is specified.
		const = <section name> |	Constant area section name is specified.
		data = <section name |	Initialized data area section name is specified.
		bss =<section name>	Non-initialized data area section name is specified.
		Default: **p = P, c = C, d = D,b = B**	
Help message	help*6		Output

Notes: *1. **show** option is invalid when **listfile** is specified.

*2. The assignments of **show = width = 0** or **show = length = 0** are interpreted as below.
show = width = 0: No line feed is performed until line feed code is output.
show = length = 0: Maximum line number is not specified, and page feed is not performed.

*3. If file name is not specified, standard file extension is added to the source file name.

*4. Macro names specified by options are shown in table 3-3.

Table 3-3 Macro Names, Names, and Constants Specified by the define Option

Item	Explanation
Macro name	A character string beginning with an alphabetic letter or an underscore followed by zero or more alphabetic letters, underscores, and numbers (0 to 9).
Name	A character string beginning with a letter or an underscore followed by zero or more alphabetic letters, underscores, and numbers.
Constant	A character string of one or more numbers, or a character string of one or more numbers followed by a period (.) and zero or more numbers.

*5. Refer to descriptions in Preprocessor Specifications,in Appendix A.1 for details on how to retrieve the include file.

*6. When the **help** option is specified, all other options are invalid.

3.4 Option Combinations

If a pair of conflicting options or suboptions are specified for a file, only one of them is considered valid. Table 3-4 shows such option combinations.

Table 3-4 Option Combinations

Option Combinations	
Valid Option	**Invalid Options**
nolistfile	**show**
code = asmcode	**debug**, and **show = object**
help	All other options

3.5 C Compiler Listings

This section describes C compiler listings and their formats.

Structure of C Compiler Listings: Table 3-5 shows the structure and contents of C compiler listings.

Table 3-5 Structure and Contents of C Compiler Listings

List Structure	Contents	Option Specification Method *1	Default
Source listing	Listing consists of source programs	**show = source** **show = nosource**	Output
	Source program listing after include file and macro expansion	**(show = include)** *2 **(show = expansion)** **(show = noinclude)** **(show = noexpansion)**	No output
Object listing	Machine language generated by the C compiler	**show = object** **show = noobject**	Output
Statistics	Total number of errors, the number of source program lines, length of each section (byte), and the number of symbols	**show = statistics** **show = nostatistics**	Output
command line specification	File names and options specified on the **command** line	—	Output

Notes: *1. All options are valid when **listfile** is specified.

*2. The option enclosed in parentheses is only valid when **show = source** is specified.

Source Listing: The source listing can be output in two ways. When **show = noinclude** and **show = noexpansion** is specified, the unpreprocessed source program is output. When **show = include** or **show =expansion** is specified, the preprocessed source program is output. Figures 3-1 and 3-2 show these output formats, respectively. Bold characters in figure 3-2 show the differences.

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

  Seq    File       Line      0----+----1----+----2----+----3----+----4----+----5-
    1 m0260.c          1       #include "header.h"
    4 m0260.c          2
    5 m0260.c          3       int sum2(void)
    6 m0260.c          4       {   int j;
    7 m0260.c          5
    8 m0260.c          6       #ifdef SMALL
    9 m0260.c          7           j=SML_INT;
   10 m0260.c          8       #else
   11 m0260.c          9           j=LRG_INT;
   12 m0260.c         10       #endif
   13 m0260.c         11
   14 m0260.c         12           return j; /* continue12345678901234567890123456
    ①    ②           ③      ±2345678901234567890  */
   15 m0260.c         13      ⑦}
```

Figure 3-1 Source Listing Output for show = noinclude and noexpansion

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

  Seq    File       Line      0----+----1----+----2----+----3----+----4----+----5-
    1 m0260.c          1       #include "header.h"
    2 header.h         1       #define SML_INT          1   }④
    3 header.h         2       #define LRG_INT        100   }
    4 m0260.c          2
    5 m0260.c          3       int sum2(void)
    6 m0260.c          4       {   int j;
    7 m0260.c          5
    8 m0260.c          6       #ifdef SMALL
    9 m0260.c          7     X     j=SML_INT;
   10 m0260.c          8     ⑤ #else
   11 m0260.c          9     E     j=100;
   12 m0260.c         10     ⑥ #endif
   13 m0260.c         11
   14 m0260.c         12           return j; /* continue12345678901234567890123456
    ①    ②           ③      ±2345678901234567890  */
   15 m0260.c         13      } ⑦
```

Figure 3-2 Source Listing Output for show = include and expansion

Description

① Listing line number

② Source program file name or include file name

③ Line number in source program or include file

④ Source program lines resulting from an include file expansion when **show = include** is specified.

⑤ Source program lines that are not to be compiled due to conditional directives such as **#ifdef** and **#elif** are marked with an X when **show=expansion** is specified.

⑥ Lines containing a macro expansion due to **#define** directives are marked with an E when **show=expansion** is specified.

⑦ If a source program line is longer than the maximum listing line, the continuation symbol (+) is used to indicate that the source program line is extended over two or more listing lines.

Object Listing: Figure 3-3 shows an example of an object listing.

```
************ SOURCE LISTING ************

FILE NAME: m0251.c

  Seq File          Line     0----+----1----+----2----+----3----+----4----+----5-
    1 m0251.c          1      extern int sum(int);
    2 m0251.c          2
    3 m0251.c          3      int
    4 m0251.c          4      sum(int x)
    5 m0251.c          5      {
    6 m0251.c          6              int i;
    7 m0251.c          7              int j;
    8 m0251.c          8
    9 m0251.c          9              j=0;
   10 m0251.c         10              for(i=0; i<=x; i++) {
   11 m0251.c         11                      j+=i;
   12 m0251.c         12              }
   13 m0251.c         13              return j;
   14 m0251.c         14      }

************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT     OFFSET   CODE          C LABEL     INSTRUCTION OPERAND      COMMENT
①       ②        ③                                ④                ⑤
P                                          ; File m0251.c    , Line 4     ; block
        00000000                           _sum:                  ⑥      ; function: sum
                                                                          ; frame size=8 ⑦
        00000000   7FF8                            ADD       #-8,R15
                                           ; File m0251.c    , Line 5     ; block
                                           ; File m0251.c    , Line 9     ; expression statement
        00000002   E300                            MOV       #0,R3
        00000004   2F32                            MOV.L     R3,@R15
                                           ; File m0251.c    , Line 10    ; for
        00000006   E300                            MOV       #0,R3
        00000008   1F31                            MOV.L     R3,@(4,R15)
        0000000A   A009                            BRA       L104
        0000000C   0009                            NOP
        0000000E                           L105:
                                           ; File m0251.c    , Line 10    ; block
                                           ; File m0251.c    , Line 11    ; expression statement
        0000000E   52F1                            MOV.L     @(4,R15),R2
        00000010   61F3                            MOV       R15,R1
              :                                          :
```

Figure 3-3 Object Listing

Description

① Section attribute (P, C, D, B) of each section
② The offset indicates the offset address relative to the beginning of each section.
③ Contents of the offset address of each section
④ Assembly code corresponding to machine language
⑤ Comments indicating the C program structure (only output when not optimized; however, labels are always output)
⑥ Line information corresponding to the C program (only output when not optimized)
⑦ Stack frame size in bytes (always output)

Statistics Information: Figure 3-4 shows an example of statistics information.

```
******** STATISTICS INFORMATION ********

********** ERROR INFORMATION ***********

NUMBER OF ERRORS:              0                    } ①
NUMBER OF WARNINGS:            0

******* SOURCE LINE INFORMATION ********

COMPILED SOURCE LINE:          13                   } ②

******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P): 0x00004A Byte(s)
CONSTANT SECTION(C): 0x000000 Byte(s)               } ③
DATA     SECTION(D): 0x000000 Byte(s)
BSS      SECTION(B): 0x000000 Byte(s)

 TOTAL PROGRAM SIZE: 0x00004A Byte(s)


********** LABEL INFORMATION ***********

NUMBER OF EXTERNAL REFERENCE  SYMBOLS:        0     } ④
NUMBER OF EXTERNAL DEFINITION SYMBOLS:        1
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:          4
```

Figure 3-4 Statistics Information

Description

① Total number of messages by the level

② Number of compiled lines from the source file

③ Size of each section and total size of sections

④ Number of external reference symbols, number of external definition symbols, and total number of internal and external labels

Note: Section size information (③) and label information (④) are not output if an error-level error or a fatal-level error has occurred when option **noobject** is specified. In addition, section size information (③) is not output when option **code = asmcode** is specified.

command Line Specification: The file names and options specified on the **command** line when the compiler is invoked are displayed. Figure 3-5 shows an example of **command** line specification information.

```
*** COMMAND PARAMETER ***

-listfile test.c
```

Figure 3-5 command Line Specification

PART II

PROGRAMMING

Section 1 Limitations of the C compiler

Table 1-1 shows the limits on source programs that can be handled by the C compiler. Source programs must fall within these limits. To edit and compile efficiently, it is recommended to split the source program into smaller programs (approximately 2 ksteps) and compile them separately.

Table 1-1 Limitation of the C Compiler

		Limit	
Classification	**Item**	**UNIX**	**PC**
Invoking the C compiler	Number of source programs that can be compiled at one time	16	1
	Total number of macro names that can be specified using the **define** option	16	16
	Length of file name (characters)	128	128
Source programs	Length of one line (characters)	4096	512
	Number of source program lines	32767	16383
Preprocessing	Nesting level of files in an **#include** directive	8	5
	Total number of macro names that can be specified in a **#define** directive *1	4096	1024
	Number of arguments that can be specified using a macro definition or a macro call operation	63	31
	Depth of the recursive expansion of a macro name	32	16
	Nesting level of **#if**, **#ifdef**, **#ifndef**, **#else**, or **#elif** directives	32	6
	Total number of operators and operands that can be specified in an **#if** or **#elif** directive	512	210
Declarations	Number of function definitions	512	256
	Number of external identifiers used for external linkage*2	4096	511
	Number of internal identifiers that can be used in one function	4096	512
	Number of internal labels*3	16384	2048
	Number of symbol table entries*4	8192	1024
	Total number of pointers, arrays, and functions that qualify the basic type	16	16
	Array dimensions	6	6

Table 1-1 Limitation of the C Compiler (cont)

Classification	Item	Limit	
		UNIX	PC
Statements	Nesting levels of compound statements	32	15
	Levels of statement nesting in a combination of repeat (**while**, **do**, and **for**) and select (**if** and **switch**) statements	32	15
	Number of **goto** labels that can be specified in one function	511	256
	Number of **switch** statements	256	128
	Nesting levels of **switch** statements	16	15
	Number of **case** labels	511	255
	Nesting levels of **for** statements	16	15
Expressions	Number of arguments that can be specified using a function definition or a function call operation	63	31
	Total number of operators and operands that can be specified in one expression	About 500	About 200
C library functions	Number of files that can be opened simultaneously by the **open** function	20	20

Notes: *1. As the C compiler itself defines five macro names (__LINE__, __FILE__, __DATE__, __TIME__, and __STDC__), the user can define a maximum of 4091 macro names in UNIX systems and a maximum of 1019 macro names in PC systems.

*2. As the C compiler itself defines two symbols, the user can define a maximum of 4094 external identifiers in UNIX systems and a maximum of 509 external identifiers in PC systems.

*3. An internal label is internally generated by the C compiler to indicate a static variable address, **case** label address, **goto** label address, or a branch destination address generated by **if**, **switch**, **while**, **for**, and **do** statements.

*4. The number of symbol table entries is determined by adding the following numbers:
Number of external identifiers
Number of internal identifiers for each function
Number of string literals
Number of initial values for structures and arrays in compound statements
Number of compound statements
Number of **case** labels
Number of **goto** labels

Section 2 Executing a C Program

This section covers object programs which are generated by the C compiler. In particular, this section explains what items are required to link C programs with assembly programs and how to install programs on the SH system (see PART III, SYSTEM INSTALLATION). This section consists of the following three parts.

Section 2.1 Structure of Object Programs

This section discusses the characteristics of memory areas used for C source programs and standard library functions.

Section 2.2 Internal Data Representation

This section explains the internal representation of data used by a C program. This information is required when data is shared among C programs, hardware, and assembly programs.

Section 2.3 Linkage with Assembly Programs

This section explains the rules for variable and function names that can be mutually referenced by multiple object programs. This section also discusses how to use registers, and how to transfer arguments and return values when a C program calls a function. The above information is required for C program functions calling assembly program routines or assembly program routines calling C program functions.

Refer to respective hardware manuals for details on SH hardware.

2.1 Structure of Object Programs

This section explains the characteristics of memory areas used by a C program or standard library function in terms of the following items.

① Sections
Composed of memory areas which are allocated statically by the C compiler. Each section has a name and type. A section name can be changed by the compiler option **section**.

② Write Operation
Indicates whether write operations are enabled at program execution.

③ Initial Value
Shows whether there is an initial value when program execution starts.

④ Alignment
Restricts addresses to which data is allocated.

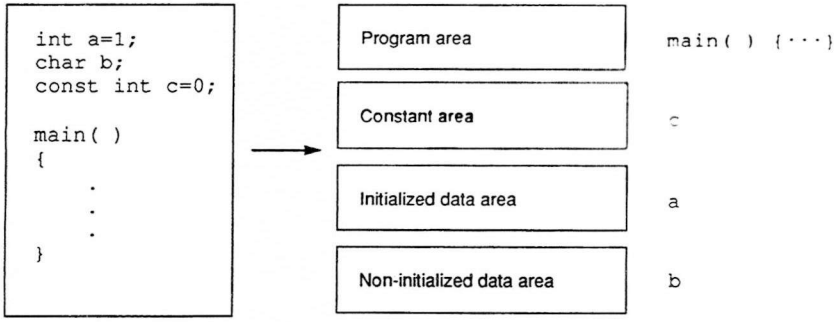
Table 2-1 shows the types and characteristics of those memory areas.

Table 2-1 Memory Area Types and Characteristics

Memory Area Name	Section Name *	Section Type	Write Operation	Initial Value	Alignment	Contents
Program area	P	code	Disabled	Yes	4 bytes	This area stores machine codes.
Constant area	C	data	Disabled	Yes	4 bytes	This area stores **const** data.
Initialized data area	D	data	Enabled	Yes	4 bytes	This area stores data whose initial values are specified.
Non-initialized data area	B	data	Enabled	No	4 bytes	This area stores data whose initial values are not specified.
Stack area	—	—	Enabled	No	4 bytes	This area is allocated at run time and is required for C program execution. Refer to section 2.2, Dynamic Area Allocation, in PART III, SYSTEM INSTALLATION.
Heap area	—	—	Enabled	No	—	This area is used by a C library function (**malloc**, **realloc**, or **calloc**). Refer to section 2.2, Allocation to Dynamic Area, in PART III, SYSTEM INSTALLATION.

Note: * Section name shown is the default generated by the C compiler when a specific name is not specified by the compiler option **section**.

Example: This program example shows the relationship between a C program and the sections generated by the C compiler.



C program

Section generated by the C compiler

2.2 Internal Data Representation

This section explains the internal representation of C language data types. The internal representation of data is determined according to the following four items:

① Size
Shows the amount of memory needed to store the data.

② Alignment
Restricts the addresses to which data is allocated. There are three types of alignment, 1-byte alignment in which data can be allocated to any address, 2-byte alignment in which data is allocated to an even byte address, and 4-byte alignment in which data is allocated to an address indivisible by four.

③ Data range
Shows the range of scalar-type data.

④ Data allocation example
Shows how the elements of aggregate-type data are allocated.

Scalar-Type Data: Table 2-2 shows the internal representation of scalar-type data used in C.

Table 2-2 Internal Representation of Scalar-Type Data

				Data Range	
Data Type	Size (bytes)	Alignment (bytes)	Sign Bit	Minimum Value	Maximum Value
char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
signed char	1	1	Used	-2^7 (–128)	$2^7 - 1$ (127)
unsigned char	1	1	Unused	0	$2^8 - 1$ (255)
short	2	2	Used	-2^{15} (–32768)	$2^{15} - 1$ (32767)
unsigned short	2	2	Unused	0	$2^{16} - 1$ (65535)
int	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
unsigned int	4	4	Unused	0	$2^{32} - 1$ (4294967295)
long	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
unsigned long	4	4	Unused	0	$2^{32} - 1$ (4294967295)
enum	4	4	Used	-2^{31} (–2147483648)	$2^{31} - 1$ (2147483647)
float	4	4	Used	$-\infty$	$+\infty$
double long double	8	4	Used	$-\infty$	$+\infty$
Pointer	4	4	Unused	0	$2^{32} - 1$ (4294967295)

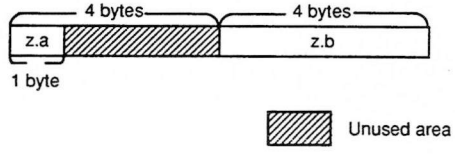
Aggregate-Type Data: This part explains the internal representation of array, structure, and union data types. Table 2-3 shows the internal data representation of aggregate-type data.

Table 2-3 Internal Representation of Aggregate-Type Data

Data Type	Alignment (bytes)	Size (bytes)	Data Allocation Example
Array type	Array element alignment	(Number of array elements) x (Element size)	`int a[10];` Alignment: 4 bytes Size: 40 bytes
Structure type	Maximum structure member alignment	Total member size *1	`struct {` `int a, b;` `};` Alignment: 4 bytes Size: 8 bytes
Union type	Maximum union member alignment	Maximum value of member size *2	`union {` `int a,b;` `};` Alignment: 4 bytes Size: 4 bytes

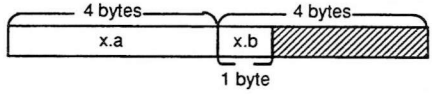
Notes: *1. When structure members are allocated, unused area may be generated between structure members to align data types.

```
struct {
    char a;
    int b;}z;
```



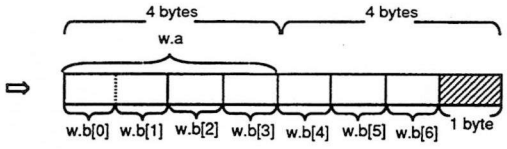
If a structure has 4-byte alignment and the last member ends at an address indivisible by four, the remaining bytes are included in this structure.

```
struct {
    int a;
    char b;}x
```



*2. When an union has 4-byte alignment and the maximum size of its members is not a multiple of four, the remaining bytes up to a multiple of four are included in this union.

```
union {
    int a;
    char b[7];}w
```



Bit Fields: A bit field is a member of a structure. This part explains how bit fields are allocated.

- Bit field members

Table 2-4 shows the specifications of bit field members.

Table 2-4 Bit Field Member Specifications

Item	Specifications
Type specifiers allowed for bit fields	**char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, and **unsigned long**
How to treat a sign when data is expanded to the declared type *1	A bit field with no sign (**unsigned** type is specified): Zero extension *2 A bit field with a sign (**unsigned** is not specified): Sign extension *2

Notes: *1. To use a member of a bit field, data in the bit field is expanded to the declared type.

*2. Zero extension: Zeros are written to the high order bits during extension.
Sign extension: The most significant bit of a bit field is used as a sign and is written to all higher-order bits generated during data extension.

Note: One-bit field data with a sign is interpreted as the sign, and can only indicate 0 and –1. To indicate 0 and 1, bit field data must be declared with **unsigned**.

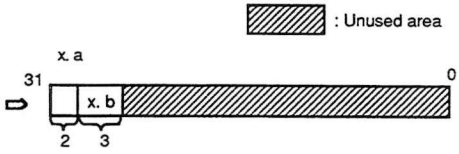
- Bit field allocation

  Bit field members are allocated according to the following five rules:

  ① Bit field members are placed in an area beginning from the left, that is, the most significant bit.

Example:

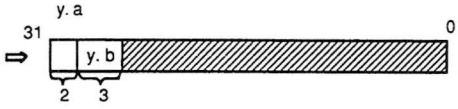
```
struct b1{
        int a:2;
        int b:3;
    }x;
```



② Consecutive bit field members having type specifiers of the same size are placed in the same area as much as possible.

Example:

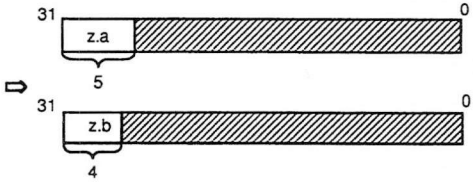
```
struct b1{
        long          a:2;
        unsigned int b:3;
    }y;
```



③ Bit field members having type specifiers with different sizes are allocated to different areas.

Example:

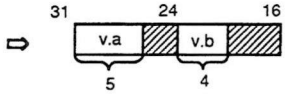
```
struct b1{
        int  a:5;
        char b:4;
    }z;
```



④ If the number of remaining bits in the area is less than the next bit field size, though type specifiers indicate the same size, the remaining area is not used and the next bit field is allocated to the next area.

Example:

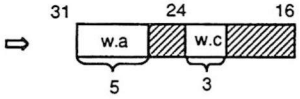
```
struct b2{
        char a:5;
        char b:4;
  }v;
```



⑤ If an anonymous bit field member or a bit field member with a bit field size of 0 is declared, the next member is allocated to the next area.

Example:

```
struct b2{
        char a:5;
        char  :0;
        char c:3;
  }w;
```



2.3 Linkage with Assembly Programs

Because C is suitable for writing system programs, it can be used to describe almost all processes in microcomputer application systems. In particular, the SH-series C compiler supports operations, such as access to the SH microcomputer registers as intrinsic functions. Refer to section 3.2, Intrinsic Functions, in Part II, Programming, for details on intrinsic functions.

Processes which cannot be written in C, for example, calculations like multiplication and addition performed by the MAC instruction, must be written in assembly language, and then linked with the C program.

This section explains two key items which must be considered when linking a C program to an assembly program:

- External identifier reference
- Function call interface

2.3.1 External Identifier Reference

Functions and variable names declared as external identifiers in a C program can be referenced or modified by both assembly programs and C programs. The following are regarded as external identifiers by the C compiler:

- A global variable which has a storage class other than **static**
- A variable name declared in a function with storage class **extern**
- A function name whose storage class is other than **static**

When variable or function names which are defined as external identifiers in C programs, are used in assembly programs, an underscore character (_) must be added at the beginning of the variable or function name (up to 31 characters without the leading underscore).

Example 1: An external identifier defined in an assembly program is referenced by a C program

- In an assembly program, symbol names beginning with an underscore character (_) are declared as external identifiers by an .EXPORT directive.
- In a C program, symbol names (with no underscore character (_) at the head) are declared as external identifiers.

Assembly program (definition)

```
    .EXPORT   _a, _b
    .SECTION D,DATA,ALIGN=4
_a: .DATA.L   1
_b: .DATA.L   1
    .END
```

C program (reference)

```
extern int a,b;


f()
{
        a+=b;
}
```

Example 2: An external identifier defined in a C program is referenced by an assembly program

- In a C program, symbol names (with no underscore character (_) at the head) are defined as external identifiers.
- In an assembly program, external references to symbol names beginning with an underscore character (_) are declared by an .IMPORT directive.

C program (definition)

```
int a;
```

Assembly program (reference)

```
      .IMPORT    _a
      .SECTION   P,CODE,ALIGN=2
      MOV.L      A_a,R1
      MOV.L      @R1,R0
      ADD        #1,R0
      RTS
      MOV.L      R0,@R1
      .ALIGN     4
A_a:  .DATA.L    _a
      .END
```

2.3.2 Function Call Interface

When either a C program or an assembly program calls the other, the assembly programs must be created using rules involving the following:

(1) Stack Pointer
(2) Allocating and Deallocating Stack Frames
(3) Registers
(4) Setting and Referencing Parameters and Return Values

Stack Pointer: Valid data must not be stored in a stack area with an address lower than the stack pointer, since the data may be destroyed by an interrupt process.

Allocating and Deallocating Stack Frames: In a function call (right after the JSR or the BSR instruction has been executed), the stack pointer indicates the lowest address of the stack used by the calling function. Allocating and setting data at addresses greater than this one is a role of the calling function. After the called function deallocates the area it has set with data, control returns to the calling function usually with the RTS instruction. The calling side then deallocates the area having an address higher than the return value address and the parameter area.

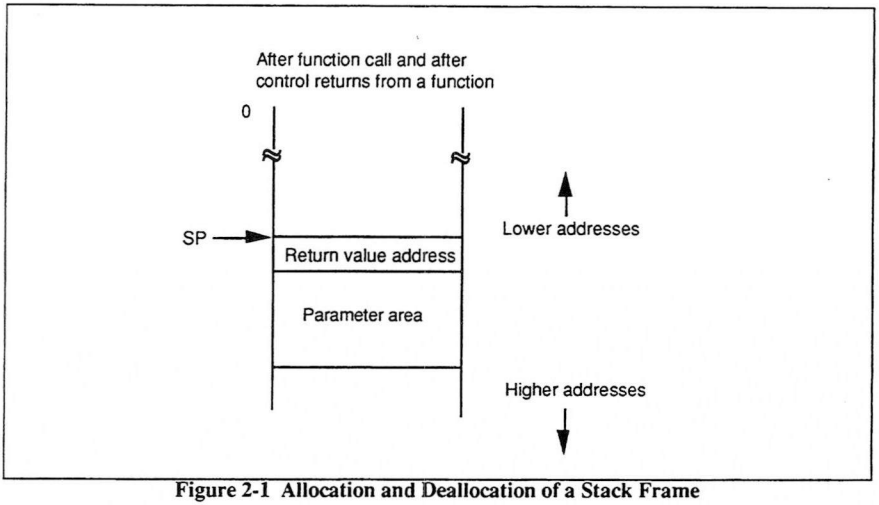


Figure 2-1 Allocation and Deallocation of a Stack Frame

Registers: Some registers change after a function call, while some do not. Table 2-5 shows how registers change according to the rules.

Table 2-5 Rules on Changes in Registers After a Function Call

Item	Registers Used in a Function	Notes on Programming
Guaranteed registers	R0 – R7	If registers used in a function contain valid data when a program calls the function, the program must push the data onto the stack or register before calling the function.
Non-guaranteed	R8 – R15, MACH, MACL, and PR	The data in registers used in functions is pushed onto the stack or register before calling the function, and popped from the stack or register only after control returns from the function.

The following examples show the rules governing register changes.

(a) A subroutine in an assembly program is called by a C program

Assembly program (called program)

```
        .EXPORT  _sub
        .SECTION P,CODE,ALIGN=2
_sub:   MOV.L    R14,@-R15
        MOV.L    R13,@-R15
        ADD      #-8,R15
                 .
                 .
                 .
        ADD      #8,R15
        MOV.L    @R15+,R13
        RTS
        MOV.L    @R15+,R14
        .END
```

Data in those registers needed by the called function is pushed onto the stack.

Function processing
(Since data in registers R0 to R7 is pushed onto a stack by the calling C program, the assembly program can use them freely without having to save them first.)

Register data is popped from the stack.

C program (calling program)

```
extern void sub();
f()
{
        sub();
}
```

(b) A subroutine in a C program is called by an assembly program

C program (called program)

```
void sub()
{
      .
      .
      .
}
```

Assembly program (calling program)

```
        .IMPORT   _sub
        .SECTION P,CODE,ALIGN=2
              .
              .
              .
        STS.L     PR,@-R15

        MOV.L     R1,@(1,R15)
        MOV       R3,R12
        MOV.L     A_sub,R0
        JSR       @R0
        NOP
        LDS.L     @R15+,PR
              .
              .
              .
A_sub:  .DATA.L   _sub
        .END
```

The called function is declared by the .IMPORT control instruction with an underscore character (_) at the beginning.

Store the PR register (return address storage register) when calling the function. If registers R0 and R7 contain valid data, the data is pushed onto the stack or stored in unused registers.

The sub function is called.

The PR register is restored.

Address data of the sub function

Setting and Referencing Parameters and Return Values: This section explains how to set and reference parameters and return values. The rules for parameters and return values differ depending on whether or not the type of each parameter or return value is explicitly declared in the function declaration. A function prototype declaration is used to explicitly declare the type of each parameter or return value.

The rest of this section explains the general rules concerning parameters and return values, how the parameter area is allocated, and how areas are established for return values.

(a) General rules concerning parameters and return values

(i) Passing parameters

A function is called only after parameters have been copied to a parameter area in registers or on the stack. Since the calling function does not reference the parameter area after control returns to it, the calling function is not affected even if the called function modifies the parameters.

(ii) Rules on type conversion

Type conversion may be performed automatically when parameters are transferred or a return value is returned. This section explains the rules on type conversion.

— Type Conversion of Parameters Whose Types are Declared

Parameters whose types are declared by prototype declaration are converted to the declared types.

— Type conversion of parameters for which types are not declared

Parameters whose types are not declared by prototype declaration are converted according to the following rules:

- Parameters whose types are **char**, **unsigned char**, **short**, or **unsigned short** are converted to **int**.
- Parameters whose types are **float** are converted to **double**.
- Other parameters are not converted.

— Return value type conversion

A return value is converted to the data type returned by the function.

Example:

```
① long f( );
  long f( )
  {   float x;
      return x;
  }
```

The return value is converted to long.

```
② void p ( int,... );
  f( )
  {   char c;
      P ( 1.0, c );
  }
```

c is converted to int because a type is not declared for the parameter.

1.0 is converted to int because the type of the parameter is int.

Note: When parameter types are not declared by a prototype declaration, the correct specifications must be made by the calling and called functions so that parameters are correctly transferred. Otherwise, correct operation is not guaranteed.

Example:

```
f(x)
float x;
{
   .
   .
   .
}

main()
{
    float x;
    f(x);
}
```

Incorrect specification

```
f(float x)
{
   .
   .
   .
}

main()
{
    float x;
    f(x);
}
```

Correct specification

Since the parameter type belonging to function f is not declared by a prototype declaration in the incorrect specification above, parameter x is converted to **double** when function main calls function f. Function f cannot receive the parameter correctly because the parameter type is declared as **float** in function f. Use the prototype declaration to declare the parameter type, or make the parameter declaration **double** in function f.

The parameter type is declared by a prototype declaration in the correct specification above.

(b) Parameter area allocation

Parameters are allocated to registers, or when this is impossible, to a stack parameter area. Figure 2-2 shows the parameter area allocation. Table 2-6 lists the general parameter area allocation rules.

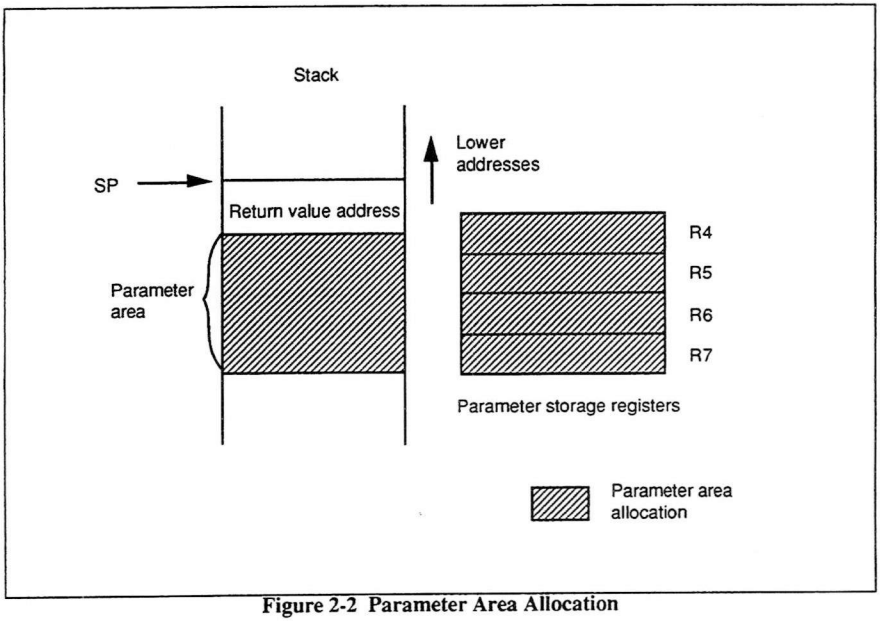


Figure 2-2 Parameter Area Allocation

Table 2-6 General Rules on Parameter Area Allocation

Allocation Rules		
Parameters Allocated to Registers		
Parameter Storage Registers	**Target Type**	**Parameters Allocated to a Stack**
R4 – R7	**char, unsigned char, short, unsigned short,int, unsigned int, long, unsigned long, float,** and pointer	① Parameters whose types are other than target types for register passing ② Parameters of a function which has been declared by a prototype declaration to have variable-number parameters* ③ Other parameters are already allocated to R4 – R7.

Note: * If a function has been declared to have variable-number parameters by a prototype definition, parameters which do not have a corresponding type in the declaration and the immediately preceding parameter are allocated to a stack.

Example:

```
int f2(int,int,int, int,...);
       :
       f2(a,b,c,x,y,z);
       :
```

→ x, y, and z are allocated to a stack.

(c) Parameter allocation

(i) Allocation to parameter storage registers

Following the order of their declaration in the source program, parameters are allocated to the parameter storage registers starting with the smallest numbered register. Figure 2-3 shows an example of parameter allocation to registers.

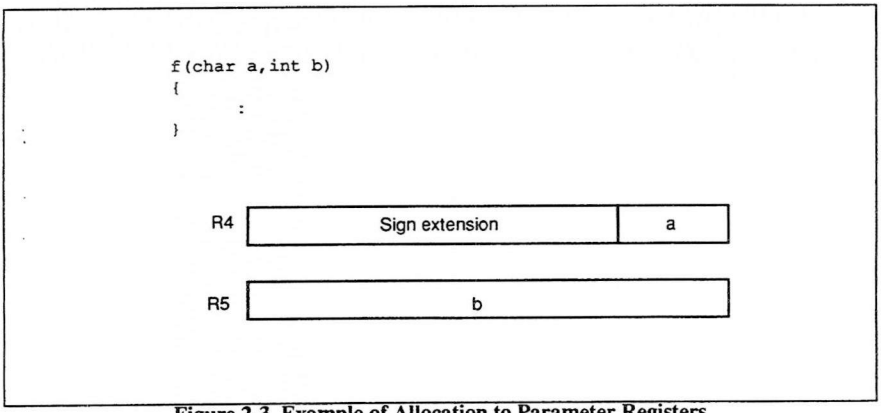


Figure 2-3 Example of Allocation to Parameter Registers

(ii) Allocation to a stack parameter area

Parameters are allocated to the stack parameter area starting from lower addresses, in the order that they are specified in the source program.

Note: Regardless of the alignment determined by the structure type, structure type or union type parameters are allocated using 4-byte alignment. Also, the area size for each parameter must be a multiple of four bytes. This is because the SH stack pointer is incremented or decremented in 4-byte units.

Refer to appendix B, Parameter Allocation Examples, for examples of parameter allocation.

(d) Return value location

The return value is written to either a register or memory depending on its type. Refer to table 2-7 for the relationship between the return value type and location.

When a function return value is to be written to memory, the return value is written to the area indicated by the return value address. The calling side must allocate this return value setting area in addition to the parameter area, and must set the address of the former in the return value address area before calling the function. The return value is not written if its type is **void**.

Table 2-7 Return Value Type and Setting Location

Return Value Type	Return Value Location
char, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, and Pointer	R0: 32 bits (If the return value type is **char** or **short**, perform sign extension before setting the return value in R0. If the return value type is **unsigned char** or **unsigned short**, perform zero extension before setting it in R0.)
double, **long double**, structure, and union	Return value setting area (memory)

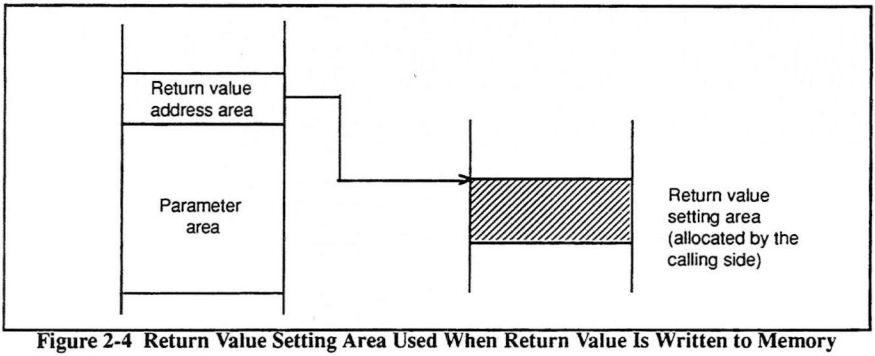


Figure 2-4 Return Value Setting Area Used When Return Value Is Written to Memory

Section 3 Extended Specifications

This section describes two C compiler extended specifications: interrupt functions and intrinsic functions.

3.1 Interrupt Functions

A preprocessor directive (**#pragma**) specifies an external (hardware) interrupt function. The following section describes how to create an interrupt function.

Description:

`#pragma interrupt` (function name [(interrupt specifications)]
[, function name [(interrupt specifications)]...])

Table 3-1 lists interrupt specifications.

Table 3-1 Interrupt Specifications

Item	Form	Options	Specifications
Stack switching	sp=	<variable> | &<variable> | <constant>	The address of a new stack is specified with a variable or a constant. <variable>: Variable (object type) value &<variable>: Variable (pointer type) address <constant>: Constant value
Trap-instruction return	tn=	<constant>	Termination is specified by the TRAPA instruction <constant>: Constant value (trap vector number)

Explanation: #pragma interrupt declares an interrupt function. An interrupt function will preserve register values before processing (all registers used by the function are pushed onto and popped from the stack when entering and exiting the function). The RTE instruction directs the function to return. However, if the trap-instruction return is specified, the TRAPA instruction is executed at the end of the function. An interrupt function with no specifications is processed in the usual procedure. The stack switching specification and the trap-instruction return specification can be specified together.

Example:

```
extern int  STK[100];

int  *ptr = STK + 100;
#pragma interrupt ( f(sp=ptr, tn=10) )
                        ①       ②
```

Explanation:

① Stack switching specification: ptr is set as the stack pointer used by interrupt function f.

② Trap-instruction return specification: After the interrupt function has completed its processing, TRAPA #10 is executed. The SP at the beginning of trap exception processing shown in the figure below. After the previous PC and SR (status register) are popped from the stack by the RTE instruction in the trap routine, control is returned from the interrupt function.

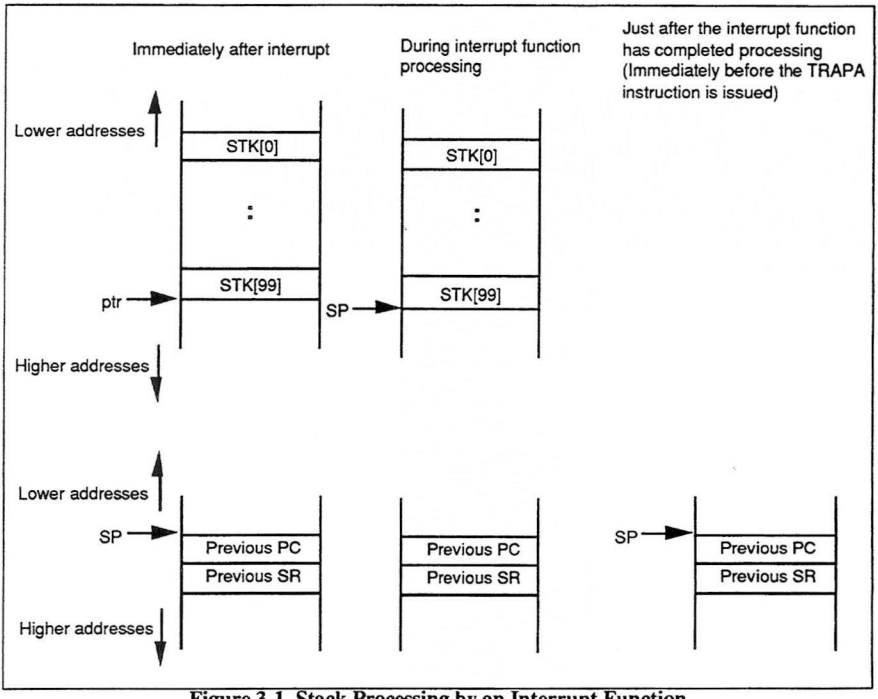


Figure 3-1 Stack Processing by an Interrupt Function

Note the following when using this function.

Warnings:

① The storage class specifier of the interrupt function must be **extern**. Even if storage class **static** is specified, the storage class is handled as **extern**.

The function must return **void** data. The **return** statement cannot have a return value. If attempted, an error is output.

Example:

```
#pragma interrupt(f1(SP=100),f2)
void f1(SP=100){…} ·····(a)
int f2(){…}  ·····(b)
```

Description:

(a) is declared correctly.

(b) returns data that is not **void**, thus (b) is declared incorrectly. An error is output.

② A function declared as an interrupt function cannot be called within the program. If attempted, an error is output. However, if the function is called within a program which does not declare it to be an interrupt function, an error is not output but correct program execution cannot be guaranteed.

Example (An interrupt function is declared):

```
#pragma interrupt(f1)
void f1(void){···}
int f2(){ f1();} ·····(a)
```

Description: Function f1 cannot be called in the program because it is declared as an interrupt function. An error is output at (a).

Example (An interrupt function is not declared):

```
int f2(){ f1();} ·····(b)
```

Description: Because function f1 is not declared as an interrupt function, an object for extern int f1(); is generated. If function f1 is declared as an interrupt function not to be compiled in the same file as f2, correct program execution is not guaranteed.

③ A function declared as an interrupt function cannot be referenced in the same file.

Example:

```
#pragma interrupt(f1)
main(){
          void (*a)(void);
          a=f1; ·····  (a)
}
```

Description: Since the address of interrupt function f1 cannot be referenced at (a), an error is output.

If an interrupt function is referenced to set, for example, a vector table, it must not be declared as an interrupt function in the same file.

Examples:

File with an interrupt function definition

```
#pragma interrupt(f1)
        .
        .
void f1(void)
{
        .
        .
}
```

File referencing an interrupt function

```
extern void f1(void); ···· (b)
main()
{
        void (*a)(void);
        a=f1;

}
```

Description: To reference the address of interrupt function f1 at (b), f1 is not declared as an interrupt function.

3.2 Intrinsic Functions

In this C compiler, system control instructions of the SH microcomputer can be written in C as intrinsic functions. The following describes the intrinsic functions provided.

Intrinsic Functions: The following functions can be specified by intrinsic functions.

① Setting and referencing the status register
② Setting and referencing the vector base register
③ I/O functions using the global base register
④ System instructions which do not compete with register sources in C

Description: #include <machine.h> must be specified when using intrinsic functions.

Intrinsic Function Specifications: Table 3-2 lists intrinsic functions.

Table 3-2 Intrinsic Functions

Item	Function	Specification	Description
Status register	Setting the status register	`void set_cr(int cr)`	Sets cr (32 bits) in the status register
	Referencing the status register	`int get_cr(void)`	References the status register
	Setting the interrupt mask	`void set_imask(int mask)`	Sets mask (4 bits) in the interrupt mask (4 bits)
	Referencing the interrupt mask	`int get_imask(void)`	References the interrupt mask (4 bits)
Vector base register (VBR)	Setting the vector base register	`void set_vbr(void **base)`	Sets **base (32 bits) in VBR
	Referencing the vector base register	`int **get_vbr(void)`	References VBR
Global base register (GBR)	Setting GBR	`void set_gbr(void *base)`	Sets *base (32 bits) in GBR
	Referencing GBR	`void *get_gbr(void)`	References GBR
	Referencing GBR-based byte	`unsigned char gbr_read_byte(int offset)`	References byte data (8 bits) at the address indicated by adding GBR and the offset specified
	Referencing GBR-based word	`unsigned word gbr_read_word(int offset)`	References word data (16 bits) at the address indicated by adding GBR and the offset specified

Table 3-2 Intrinsic Functions (cont)

Item	Function	Specification	Description
Global base register (GBR) (cont)	Referencing GBR-based long word	`unsigned long gbr_read_long(int offset)`	References long word data (32 bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based byte	`void gbr_write_byte( int offset, unsigned char data)`	Sets data (8bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based word	`void gbr_write_word( int offset, unsigned short data)`	Sets data (16 bits) at the address indicated by adding GBR and the offset specified
	Setting GBR-based long word	`void gbr_write_word( int offset, unsigned long data)`	Sets data (32 bits) at the address indicated by adding GBR and the offset specified
	AND of GBR base	`void gbr_and_byte( int offset, unsigned char mask)`	ANDs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	OR of GBR base	`void gbr_or_byte( int offset, unsigned char mask)`	ORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	XOR of GBR base	`void gbr_xor_byte( int offset, unsigned char mask)`	XORs mask with the byte data at the address indicated by adding GBR and the offset specified, and then stores the result at the same address
	TEST of GBR base	`void gbr_tst_byte( int offset, unsigned char mask)`	Checks if the byte data at the offset from GBR is 0 or not, and sets the result in the T bit
Special instructions	SLEEP instruction	`void sleep(void)`	Executes the SLEEP instruction
	TAS instruction	`void tas(char *addr)`	Executes TAS.B @addr
	TRAPA instruction	`int trapa(int trap_no)`	Executes TRAPA #trap_no

Warnings: The offsets and masks shown in table 3-2, Intrinsic Functions, must be constants. Also, the specification range for offsets is +255 bytes when the access size is shown in bytes, +510 bytes when the access size is shown as a word, and +1020 bytes when the access size is shown as a long word. Masks which can be specified for performing logical operations (AND, OR, XOR, or TEST) on a location relative to GBR (global base register) must be within the range of 0 to +255. As GBR is a control register whose contents are not preserved by all functions in this C compiler, take care when changing GBR settings.

Example:

```
#include <machine.h>

#define CDATA1 0
#define CDATA2 1
#define CDATA3 2
#define SDATA1 4
#define IDATA1 8
#define IDATA2 12

struct{
        char  cdata1;           /* offset  0*/
        char  cdata2;           /* offset  1*/
        char  cdata3;           /* offset  2*/
        char  sdata1;           /* offset  4*/
        char  idata1;           /* offset  8*/
        char  idata2;           /* offset 12*/
}table;

void f()
{
    set_gbr( &table);                   /* Set the start address of table to GBR */
          :
    gbr_write_byte( CDATA2, 10);        /* Set 10 to table.cdata2.               */
    gbr_write_long( IDATA2, 100);       /* Set 100 to table.idata2.              */
          :
    if(gbr_read_byte( CDATA2) != 10)    /* Reference table.cdata2.               */
          gbr_and_byte( CDATA2, 10);    /* AND 10 and table.cdata2, and set it   */
          :                             /* to table.cdata2.                      */
    gbr_or_byte( CDATA2, 0x0F);         /* OR H'0F and table.cdata2, and set it  */
          :                             /* to table.cdata2.                      */
    sleep();                            /* Expanded to the sleep instruction     */
}
```

Effective use of intrinsic functions:

① Set the start address of a structure which is allocated to memory and frequently accessed in GBR and access its members by gbr_read_byte, gbr_write_byte, etc.

② In the case of ①, byte data frequently used in logical operations should be declared within 128 bytes from the start address of the structure.

Section 4 Notes on Programming

This section contains notes on coding programs for the C compiler and on troubleshooting when compiling or debugging programs.

4.1 Coding Notes

Functions with float Parameters: For a function that declares **float** for parameters, either a prototype must be declared or parameters must be declared as **double**. Correct processing is not guaranteed if a function that has **float** parameters is called without a prototype declaration.

Example:

```
void f(float); ·····①

g( )
{
    float a;
    f(a);
}

void
f(float x)
{

}
```

Since function f has a **float** parameter, a prototype must be declared as shown at ①.

Program Whose Evaluation Order is Not Regulated: The effect of the execution is not guaranteed in a program whose execution results differ depending on the evaluation order.

Example:

`a[i]=a[++i];` ···· The value of i on the left side differs depending on whether the right side of the assignment expression is evaluated first.

`sub(++i, i);` ···· The value of i for the second parameter differs depending on whether the first function parameter is evaluated first.

Overflow Operation and Zero Division: At run time if overflow operation or zero division is performed, error messages will not be output. However, if an overflow operation or zero division is included in the operations for one or more constants, error messages will be output at compilation.

Example:

```
main()
{
    int ia;
    int ib;
    float fa;
    float fb;

    ib=32767;
    fb=3.4e+38f;

/* Compilation error messages are output when an overflow operation and */
/* zero division are included in operations for one or more constants.  */

    ia=99999999999; /* (W) Detect integer constant overflow. */
    fa=3.5e+40f;    /* (W) Detect floating pointing constant overflow. */
    ia=1/0;         /* (E) Detect division by zero. */
    fa=1.0/0.0;     /* (W) Detect division by floating point zero. */

/* No error message on overflow at execution is output. */

    ib=ib+32767;    /* Ignore integer constant overflow. */
    fb=fb+3.4e+38f; /* Ignore floating point constant overflow. */

}
```

Assignment to const Variables: Even if a variable is declared with **const** attribute, if assignment is done to a variable other than **const** converted from **const** attribute or if a program compiled separately uses a parameter of a different type, the C compiler cannot detect the error.

Example:

①
```
const char *p;          /* Because the first parameter p in library*/
      .                 /* function strcat is a pointer for char,  */
      .                 /* the area indicated by the parameter p   */
  strcat(p, "abc")      /* may change.                             */
```

② file 1
```
const int i;
```

file 2
```
extern int i;          /* In file 2, parameter i is not declared as   */
   :                   /* const, therefore assignment to it in file 2 */
  i=10;                /* is not an error.                            */
```

4.2 Notes on Programming Development

Table 4-1 shows troubleshootings for developing programs at compilation or when debugging.

Table 4-1 Troubleshooting

Trouble	Check Points	Solution	References
Error 314, **cannot found section**, is output at linkage	The section name which is output by the C compiler must be specified in capitals in **start** option of linkage editor.	Specify the correct section name.	Part II, Programming, 2.1
Error 105, **undefined external symbol**, is output at linkage	If identifiers are mutually referenced by a C program and an assembly program, an underscore must be attached to the symbol in the assembly program.	Reference parameters with the correct parameters.	Part II, Programming, 2.3.1
	Check if the C program uses a library function.	Specify a standard library as the input library at linkage.	Standard library specification: Part II, Programming, 4.2.1 (3) Execution routine in a standard library: Part III, System Installation, 2.1 (2)
	An undefined reference symbol identifier must not start with a _ _ (A run time routine in a standard library must be used.)		
	Check if a standard I/O library function is used in the C program.	Create low level interface routines for linking.	Part III, System Installation, 4. (6)
Debugging at the C source level cannot be performed	**debug** option must be specified at both compilation and linkage.	Specify **debug** option at both compilation and linkage.	Part I, Overview and Operation, 3.3
	A linkage editor of Ver.5.0 or higher must be used.	Use a linkage editor of Ver.5.0 or higher.	

PART III

SYSTEM INSTALLATION

Section 1 Overview

Part III describes how to install object programs generated by the C compiler on a SH system. Before installation, memory allocation and execution environment for the object program must be specified.

- Memory allocation

  Stack area, heap area, each section of a C-compiler-generated object program must be allocated in ROM or RAM on a SH system.

- Execution environment setting for a C-compiler-generated object program

  The execution environment can be specified by the register initialization processing, memory area initialization, and C program initiation processing. These must be written by assembly language.

  If C library functions for I/O are used, library must be initialized according to the execution environment specification. Specifically, if I/O function (**stdio.h**) and memory allocation function (**stdlib.h**) are used, the user must create low-level I/O routines and memory allocation routines appropriate to the user system.

Section 2 describes how to allocate C programs in memory area and how to specify linkage editor's commands that actually allocate a program in memory area, using examples.

Section 3 describes items to be specified in execution environment setting and execution environment specification programs.

Section 4 describes how to create C-library function initialization and low-level routines.

Section 2 Allocating Memory Areas

To install an object program generated by the C compiler on a system, the size of each memory area must be determined, then the areas must be appropriately allocated in memory.

Some memory areas, such as the area used to store machine code and the area used to store data declared using external definitions, are allocated statically. Other memory areas, such as the stack area, are allocated dynamically.

This section describes how the size of each area is determined and how to allocate an area in memory.

2.1 Static Area Allocation

2.1.1 Data to be Allocated in Static Area

Sections of object programs such as program area, constant area, initialized data area, and non-initialized data area are allocated to the static area.

2.1.2 Static Area Size Calculation

The static area size is calculated by adding the size of C-compiler-generated object program and that of library functions used by the C program. After object program linkage, the static area size can be determined from each section size including library size output on a linkage map listing. Before object program linkage, the static area size can be approximately determined from the section size information on a compile listing. Figure 2-1 shows an example of section size information.

```
******* SECTION SIZE INFORMATION *******

PROGRAM    SECTION(P): 0x00004A Byte(s)
CONSTANT   SECTION(C): 0x000018 Byte(s)
DATA       SECTION(D): 0x000004 Byte(s)
BSS        SECTION(B): 0x000004 Byte(s)

  TOTAL PROGRAM SIZE: 0x00006A Byte(s)
```

Figure 2-1 Section Size Information

If the standard library is not used, the static area size can be calculated by adding memory area size used by sections to the size shown in section size information. However, if the standard library is used, the memory area used by the library functions must be added to the the memory area size of each section. The standard library includes C library functions based on C language specifications and arithmetic routines required for C program execution. Accordingly, the standard library must be linked even if library functions are not used in the C source program.

For details on memory area size used by the standard library functions, refer to the attached Standard Library Memory Stack Size Listing. The following example shows how to calculate static area size based on the section size information shown in figure 2-1.

Calculation Example

<ctype.h>

Function Name	Low-Level Routine	Library *1	Memory Size (Bytes) Section P	Section B	Section C	Section D	Stack Size (Bytes)
isalnum	None	isalnum, _ctype	32	0	256	0	16
isalpha	None	isalpha, _ctype	32	0	256	0	16

Note: *1. Library functions required for linkage. The library functions include those used by the C program and the library function itself.

1. isalnum function of **<ctype.h>** is used

   Add 32 bytes to section P and 256 bytes to section C.

Section Name	Size (Bytes) C Program	Library	Total
P	74	32	106
B	24	0	24
C	4	256	260
D	4	0	4

2. isalnum and isalpha functions of **<ctype.h>** are used

   When a library function is used by multiple functions, memory size required for the library need not to be duplicated. The following table shows memory size example, when library function **_ctype** is used by multiple functions.

<Library common routine>

Section Name	Memory Size (Bytes) Section P	Section B	Section C	Section D
_ctype	0	0	256	0

Each section size is calculated by the following formula:

	Size (Byte)				
Section Name	C Program	Library 1	Library 2	Duplicated Library	Total *1
P	74	32	32	0	138
B	24	0	0	0	24
C	4	256	256	256	260
D	4	0	0	0	4

Note: *1. Section size = C program + Library 1 + Library 2 – Duplicated library
(**isalnum**) (**isalpha**) (**_ctype**)

Note: The standard library supplied by the C compiler includes C library functions (based on C language specification), and arithmetic routines (required for C program execution). The size required for run time routines must also be added to the memory area size in the same way as C library functions.

Run time routine names used by the C programs are output as external symbols in theassembly programs generated by the C compiler (option **code = asmcode**). The user can see the run time routine names used in the C programs through the assembly program listing.

The following shows the example of C program and assembly program listings.

C program

```
f( int a, int b)
{
        a /= b;
        return a;
}
```

Assembly program output by the C compiler

```
            .IMPORT     __divls      ; An external reference definition for the run time routine
            .EXPORT     _f
            .SECTION    P,CODE,ALIGN=4
_f:
            MOV         R5, R1
            MOV.L       A_divls, R2
            JSR         @R2
            MOV         R4, R0
            RTS
            NOP
A_divls:    DATA.L      __divls
            .END
```

An external reference definition (**.IMPORT**) beginning with __ indicates a run time routine. In the above example, **__divls** is a run time routine used in the C program.

2.1.3 ROM and RAM Allocation

When allocating a program to memory, static areas must be allocated to either ROM and RAM as shown below.

Program area (section P): ROM
Constant area (section C): ROM
Non-initialized data area (section B): RAM
Initialized data area (section D): ROM, RAM (for details, refer to the following section)

2.1.4 Initialized Data Area Allocation

The initialized data area contains data with initial value. Since the C language specifications allow the user to modify initialized data in programs, the initialized data area is allocated to ROM and is copied to RAM before program execution. Therefore, the initialized data area must be allocated in both ROM and RAM.

However, if the initialized data area contains only static variables that are not modified during program execution, only a ROM area needs to be allocated.

2.1.5 Example: Memory Area Allocation and Address Specification at Program Linkage

Each program section must be addressed by the option or subcommand of the linkage editor when the absolute load module is created, as described below.

Figure 2-2 shows an example of allocating static areas.

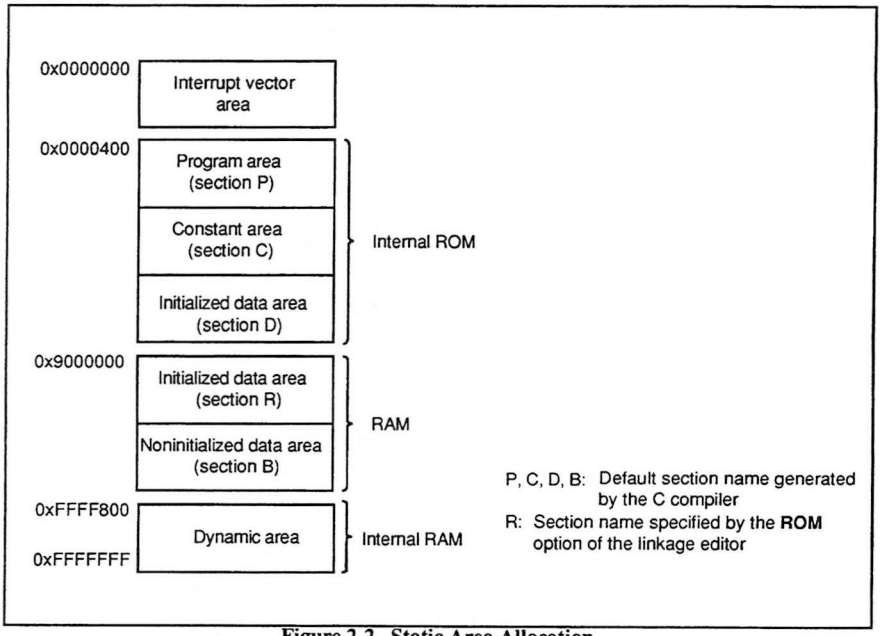


Figure 2-2 Static Area Allocation

Specify the following subcommands when allocating the static area as shown in figure 2-2.

```
      :
ROMΔ(D,R)                    ----------------①
STARTΔP,C,D(400),R,B(9000000)--------②
      :
```

Description:

① Define section R having the same size as section D, in the output load module. To reference the symbol allocated to section D, copy the contents of section D to section R and reference to the symbol in section R. Sections D and R are allocated to initialized data section in ROM and RAM, respectively.

② Allocate sections P, C, and D to internal ROM starting from address 0x400 and allocate sections R and B to RAM starting from address 0x9000000.

2.2 Dynamic Area Allocation

2.2.1 Dynamic Areas

Two types of dynamic areas are used:

① Stack area

② Heap area (used by the memory allocation library functions)

2.2.2 Dynamic Area Size Calculation

Stack Area: The stack area used in C programs is allocated each time a function is called and is deallocated each time a function is returned. The total stack area size is calculated based on the stack size used by each function and the nesting of function calls.

- Stack area used by each function

The size of stack used by each function can be determined from the object list (frame size) output by the C compiler. However, note that this does not account for the size of parameters to be pushed onto the stack when a function is called. Accordingly, the parameter size must be added to stack area size.

The following example shows the object list, stack allocation, and stack size calculation method.

Example:

The following shows the object list and stack size calculation in a C program.

```
extern int h(char, int *, double);
int
h(char a, register int *b, double c)
{
    char    *d;

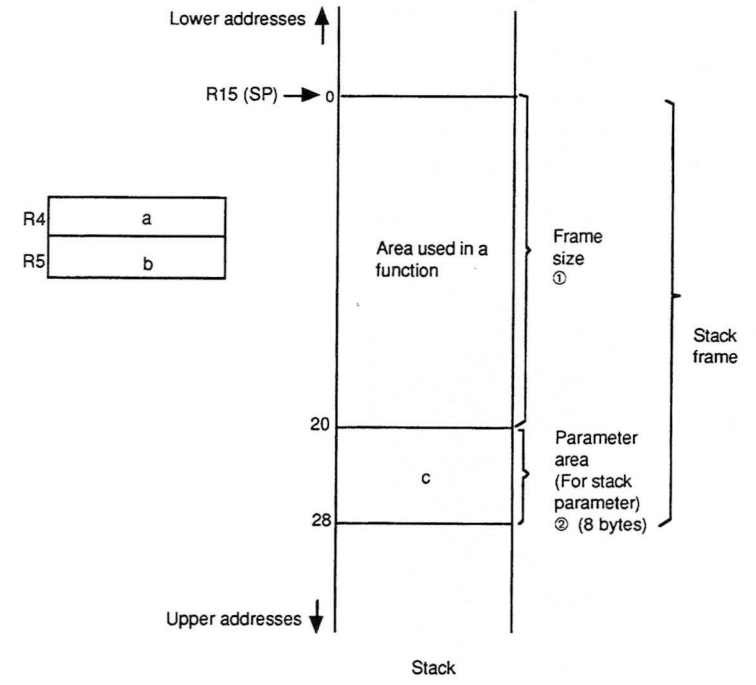
    d= &a;
    h(*d,b,c);
    {
       register int i;

       i= *d;
       return i;
    }
}
```

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT   OFFSET   CODE          C LABEL   INSTRUCTION OPERAND     COMMENT
P
      00000000                 _h:                           ; function: h
                                                             ; frame size=20
                                                                   ①
      00000000   2FE6                    MOV.L   R14,@-R15
      00000002   2FD6                    MOV.L   R13,@-R15
                                :
```



The size of stack used by a function is determined by adding frame size and parameter area size (for stack parameter). Accordingly, in the above example, the stack size used by the function is calculated as follows: 20 (①) + 8 (②) = 28 bytes

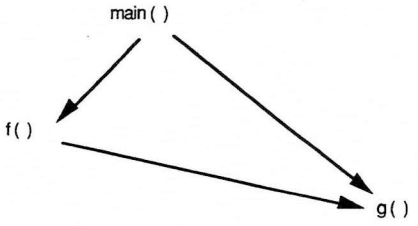
For details on the size of parameters to be pushed onto the stack, refer to the description of parameter and return value setting and referencing in section 2.3.2 of Part II.

- Stack size calculation

  The following example shows a stack size calculation depending on the function call nesting.

  Example:

  Figure 2-3 illustrates the function call nestings and stack size.



Function Name	Stack Size (Bytes)
main	24
f	32
g	24

Figure 2-3 Nested Function Calls and Stack Size

If funtion **g** is called via function **f**, stack area size is calculated according to the formula listed in table 2-1.

Table 2-1 Stack Size Calculation Example

Function Calling Route	Total Stack Size
main (24) —> f(32) —> g(24)	80 bytes (Maximum size of stack area)
main (24) —> g(24)	48 bytes

As can be seen from table 2-1, the maximum size of stack area required for the longest function calling route should be determined (80 bytes in this example) and this size of memory should be allocated in RAM.

When using standard library functions, the stack frame sizes for library functions must also be accounted for. Refer to the Standard Library Memory Stack Size Listing, included with the C compiler package.

Note: If recursive calls are used in the C source program, first determine the stack area required for a recursive call, and then multiply with the maximum number of recursive calls.

Heap Area: The total heap area required is equal to the sum of the areas to be allocated by memory management library functions (**calloc**, **malloc**, or **realloc**) in the C program. An additional 4 bytes must be summed because a 4-byte management area is used every time a memory management library function allocates an area.

An input/output library function uses memory management library functions for internal processing. The size of the area allocated in an input/output is determined by the following formula: 516 bytes x (maximum number of simultaneously open files)

Note: Areas released by the **free** function, a memory management library function, can be reused. However, since these areas are often fragmented (separated from one another), a request to allocate a new area may be rejected even if the net size of the free areas is sufficient. To prevent this, take note of the following:

① If possible, allocate the largest area first after program execution is started.

② If possible, specify data area size to be reused as a constant.

2.2.3 Rules for Allocating Dynamic Area

The dynamic area is allocated to RAM. The stack area is determined by specifying the highest address of the stack to the vector table, and refer to it as SP (stack pointer). The heap area is determined by the initial specification in the low-level interface routine (**sbrk**). For details on stack and heap areas, refer to section 3.1, Vector Table Setting (**VEC_TBL**), and section 4.6, Creating Low-Level Interface Routine, respectively.

Section 3 Setting the Execution Environment

This section describes the environment required for C program execution. A C-program environment specification program must be created according to the system specification because the C program execution environment differs depending on the user systems. In this section, basic C program execution specification, where no C library function is used, is described as an example. Refer to section 4, Setting C Library Function Execution Environment, for details on using C library functions.

Figure 3-1 shows an example of program configuration.

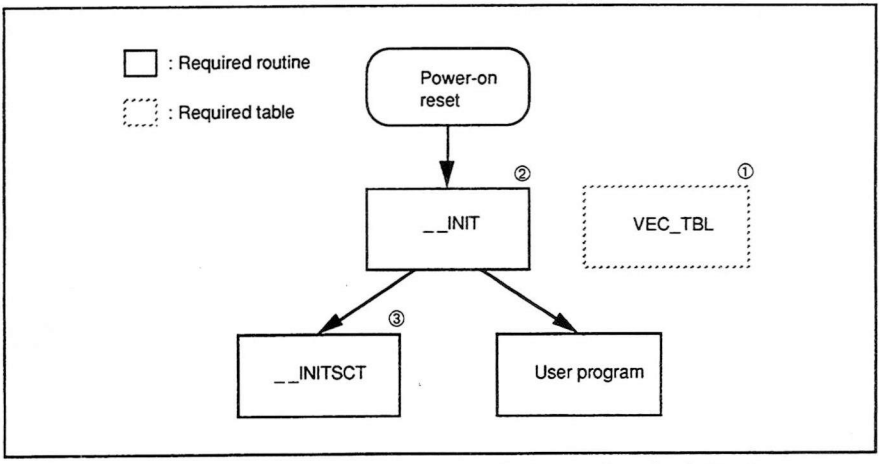


Figure 3-1 Program Configuration (No C Library Function is Used)

Each routine is described below.

① Vector table setting (**VEC_TBL**)

Sets vector table so as to initiate register initialization program **__INIT** and set the stack pointer (SP) by power-on reset.

② Initialization (**__INIT**)

Initializes registers and sequentially calls initialization routines.

③ Section initialization (**__INITSCT**)

Clears the non-initialized data area with zeros and copies the initialized data area in ROM to RAM.

How to create the above routines are described as follows.

3.1 Vector Table Setting (VEC_TBL)

To call register initialization routine **_ _INIT** at power-on reset, specify the start address of function **_ _INIT** at address 0 in the vector table. Also to specify the SP, specify the highest address of the stack to address H'4. When the user system executes interrupt handlings, interrupt vector settings are also performed in the **VEC_TBL** routine. The coding example of **VEC_TBL** is shown below.

Example:

```
.SECTION VECT,DATA, LOCATE=H'0000
                        ;   Assigns section VECT to address H'0 by the section directive.
.IMPORT    _ _INIT
.IMPORT    _IRQ0
.DATA.L    _ _INIT      ;   Assigns the start address of INIT to addresses H'0x0 to H'0x3.
.DATA.L    (a)          ;   Assigns the SP to addresses H'0x4 to H'0x7.
                        ;   (a):  The highest address of the stack
.ORG       H'00000100
.DATA.L    _IRQ0        ;   Assigns the start address of IRQ0 to addresses H'0x100 to
.END                        H'0x103.
```

3.2 Initialization (_ _INIT)

_ _INIT initializes registers, calls initialization routine sequentially, and then calls main function. The coding example of this routine is shown below.

Example:

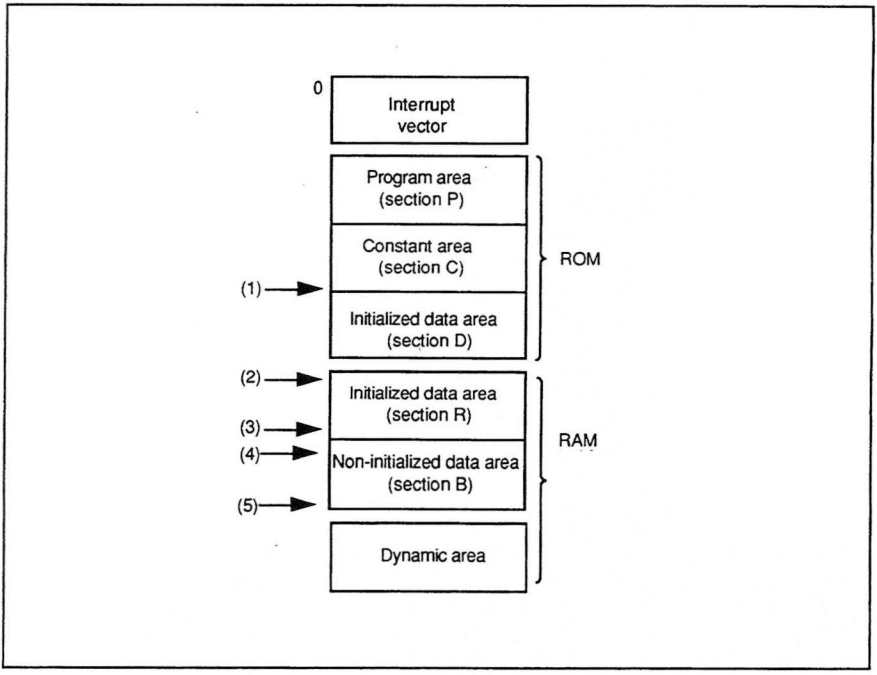
```
extern void _INITSCT(void);
extern void main(void);

void   _INIT()
{
    _INITSCT();                 /* Calls section initialization routine __INITSCT. */
    main();                     /* Calls main routine _main.                        */
    for( ;  ; )                 /* Branches to endless loop after executing main    */
        ;                       /* function and waits for reset.                    */
}
```

3.3 Section Initialization (__INITSCT)

To set the C program execution environment, clear the non-initialized data area with zeros and copy the initialized data area in ROM to RAM. To execute the **__INITSCT** function, the following addresses must be known.

- Start address (1) of initialized data area in ROM.
- Start address (2) and end address (3) of initilalized data area in RAM
- Start address (4) and end address (5) of non-initialized data area



To obtain the above addresses, create the following assembly programs and link them together.

```
            .SECTION  D,DATA,ALIGN=4
            .SECTION  R,DATA,ALIGN=4
            .SECTION  B,DATA,ALIGN=4
            .SECTION  C,DATA,ALIGN=4

__D_ROM     .DATA.L   (STARTOF D)                ; start address of section D  (1)
__D_BGN     .DATA.L   (STARTOF R)                ; start address of section R  (2)
__D_END     .DATA.L   (STARTOF R) + (SIZEOF R)   ; end address of section R    (3)
__B_BGN     .DATA.L   (STARTOF B)                ; start address of section B  (4)
__B_END     .DATA.L   (STARTOF B) + (SIZEOF B)   ; end address of section B    (5)

            .EXPORT   __D_ROM
            .EXPORT   __D_BGN
            .EXPORT   __D_END
            .EXPORT   __B_BGN
            .EXPORT   __B_END
            .END
```

Notes: ① Section names B and D must be the non-initialized data area and initialized data area section names specified with the compiler option **section**.

② Section name R must be the section name in RAM area specified with the **ROM** option at linkage.

If the above preparation is completed, section initialization routine can be written in C as shown below.

Example:

```
extern int  *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;
extern void  _INITSCT( );

void  _INITSCT( )
{
        short *p, *q ;

        /* Non-initialized area is initialized to zeros */

        for (p=_B_BGN ; p<=_B_END ; p++)
                *p=0 ;

        /* Initialized data is copied from ROM to RAM */

        for (p=_D_BGN , q=_D_ROM ; p<=_D_END ; p++, q++)
                *p=*q ;
}
```

Section 4 Setting the C Library Function Execution Environment

To use C library functions, they must be initialized to set C program execution environment. To use I/O (**stdio.h**) and memory management (**stdlib.h**) functions, low-level I/O and memory allocation routines must be created for each system.

This section describes how to set C program execution environment when C library functions are used.

Figure 4-1 shows a program configuration when C library functions are used.

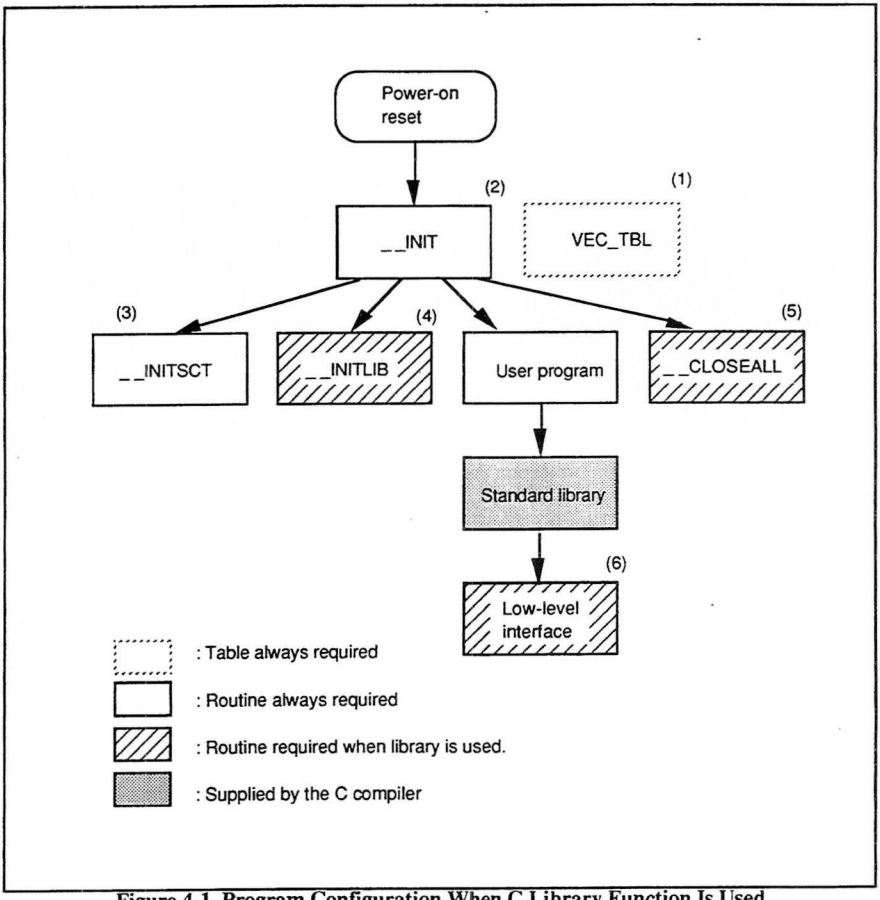


Figure 4-1 Program Configuration When C Library Function Is Used

Each routine required to execute library functions as follows.

(1) Setting vector table (**VEC_TBL**)

Sets vector table to initiate register initialization program (**_ _INIT**) and set the stack pointer (SP) at power-on reset.

(2) Initializing registers (**_ _INIT**)

Initializes registers and sequentially calls the initialization routines.

(3) Initializing sections (**_ _INITSCT**)

Clears non-initialized dasta area with zeros and copies the initialized data area in ROM to RAM. This routine is supplied as a standard library function.

(4) Initializing C library functions (**_ _INITLIB**)

Initializes C library functions required to be initialized and prepares standard I/O functions.

(5) Closing files (**_ _CLOSEALL**)

Closes all files with open status.

(6) Low-level interface routine

Interfaces library functions and user system when standard I/O and memory management library functions are used.

Creation of the above routines is described below.

Note: When using the C library functions that terminates program execution such as **exit**, **onexit**, or **abort**, the C library function must be created according to the user system. For details, refer to addpendix D, Termination Processing Function Example.

In addition, when using C library function **assert** macro, the **abort** function must be supplied.

4.1 Setting Vector Table (VEC_TBL)

Same as when no C library function is used. For details, refer to section 3, Setting the Execution Environment.

4.2 Initializing Registers (_ _INIT)

Initializes registers and sequentially calls the initialization routine _ _**INITLIB** and file closing routine _ _**CLOSEALL**. The coding example of _ _**INIT** is shown below.

```
extern void _INITSECT(void);
extern void _INITLIB(void);
extern void _CLOSEALL(void);
extern void main(void);

void _INIT(void)
{
    _INITSCT();          /* Calls section initialization routine _ _INITSCT. */
    _INITLIB();          /* Calls library initialization routine _ _INITLIB. */
    main();              /* Calls C program main function.                   */
    _CLOSEALL();         /* Calls file close routine _ _CLOSEALL.            */
    for( ; ; )           /* Branches to endless loop after executing main    */
        ;                /* function and waits for reset.                    */
}
```

4.3 Initializing Sections (__INITSCT)

Same as when the C library functions are not used. For details, refer to section 3, Setting Execution Environment.

4.4 Initializing C Library Functions (__INITLIB)

Initialization must be performed for related C library functions before being used. The following description assumes the case when the initialization is performed in **__INITLIB** in the program initiation routine.

To perform initialization, the following must be considered.

(1) **errno** indicating the library error status must be initialized for all library functions.

(2) When using each function of **<stdio.h>** and **assert** macro, standard I/O library function must be initialized.

(3) The user low-level interface routine must be initialized according to the user low-level initialization routine specification if required.

(4) When using the **rand** and **strtok** functions, library functions other than I/O must be initialized.

Library function initialization program example is shown below.

Example:

```
#include <stdlib.h>

extern void _INIT_LOWLEVEL(void) ;
extern void _INIT_IOLIB(void) ;
extern void _INIT_OTHERLIB(void) ;

void _INITLIB(void)          /*Deletes an underline from symbol name used in the assembly routine*/
{
   errno=0;                  /*Initializes library functions commonly*/

   _INIT_LOWLEVEL( ) ;       /*Calls low-level interface initialization routine*/
   _INIT_IOLIB( ) ;          /*Calls standard I/O initialization routine*/
   _INIT_OTHERLIB( ) ;       /*Calls initialization routine other than that for standard I/O*/
}
```

The following shows examples of initialization routine (**_INIT_IOLIB**) for standard I/O library function and initialization routine (**_INIT_OTHERLIB**) for other standard library function. Initialization routine (**_INIT_LOWLEVEL**) for low-level interface routine must be created according to the user low-level interface routine's specifications.

4.4.1 Creating Initialization Routine (_INIT_IOLIB) for Standard I/O Library Function

The initialization routine for standard I/O library function initializes **FILE**-type data used to reference files and open the standard I/O files. The initialization must be performed before opening the standard I/O files.

The following shows an example of **_INIT_IOLIB**.

Example:

```
#include <stdio.h>

void _INIT_IOLIB(void)
{
      FILE *fp ;

                  /*Initializes FILE-type data*/

  for (fp=_iob; fp<_iob+_NFILE; fp++){
     fp -> _bufptr=NULL ;                    /*Clears buffer pointer    */
     fp -> _bufcnt=0 ;                       /*Clears buffer counter    */
     fp -> _buflen=0 ;                       /*Clears buffer length     */
     fp -> _bufbase=NULL ;                   /*Clears base pointer      */
     fp -> _ioflag1=0 ;                      /*Clears I/O flag          */
     fp -> _ioflag2=0 ;
     fp -> _iofd=0 ;
  }

                  /*Opens standard I/O file  */

                    *1
  if (freopen( "stdin" , "r", stdin)==NULL)  /*Opens standard input file */
    stdin-> _ioflag1=0xff ;                  /*Disables file access  *2  */
    stdin-> _ioflag1 |= _IOUNBUF ;           /*No data buffering     *3  */
                    *1
  if (freopen( "stdout" , "w", stdout)==NULL)/*Opens standard output file*/
    stdout-> _ioflag1=0xff ;
  stdout-> _ioflag1 |= _IOUNBUF ;
                    *1
  if (freopen( "stderr", "w", stderr)==NULL) /*Opens standard error file */
    stderr-> _ioflag1=0xff ;
  stderr-> _ioflag1 |= _IOUNBUF ;
 }
```

Notes: *1. Standard I/O file names are specified. These names are used by the low-level interface routine **open**.
*2. If file could not be opened, the file access disable flag is set.
*3. For equipment that can be used in interactive mode such as console, the buffering disable flag is set.

```
                /*Declares FILE-type data in the C language*/

        #define _NFILE 20
        struct _iobuf{
                    unsigned char *_bufptr;   /*Buffer pointer       */
                    long           _bufcnt;   /*Buffer counter       */
                    unsigned char *_bufbase;  /*Buffer base pointer */
                    long           _buflen;   /*Buffer length       */
                    char           _ioflag1;  /*I/O flag            */
                    char           _ioflag2;  /*I/O flag            */
                    char           _iofd;     /*I/O flag            */
        }_iob[_NFILE];
```

Figure 4-2 FILE-Type Data

4.4.2 Creating Initialization Routine (_INIT_OTHERLIB) for Other Library Function

The following gives an example of the routine initializing C library functions other than standard I/O.

Example:

```
#include <stddef.h>

extern char *_slptr ;
extern void srand(unsigned int) ;

void _INIT_OTHERLIB(void)
{

   srand(1) ;          /*Sets initial value when rand function is used*/
   _slptr=NULL ;       /*Initializes the pointer used in the strtok function*/
}
```

4.5 Closing Files (_ _CLOSEALL)

When a program ends normally, all open files must be closed. Usually, the data destined for a file is stored in a memory buffer. When the buffer becomes full, data is output to an external storage device. Therefore, if the files are not closed, data remaining in buffers is not output to external storage devices and may be lost.

When an program is installed in a device, the program is not terminated normally. However, if the main function is terminated by a program error, all open files must be closed.

The following shows an example of **__CLOSEALL**.

Example:

```
#include <stdio.h>

void _CLOSEALL(void)        /*Deletes an underline from symbol name in assembly routine*/
{

      int i;

      for (i=0; i<_NFILE; i++)

            /*Checks that file is open*/

            if(_iob[i]._ioflag1 & ( _IOREAD|_IOWRITE|_IORW))

            /*Closes opened files*/

                  fclose(&_iob[i]) ;
}
```

4.6 Creating Low-Level Interface Routines

Low-level interface routines must be supplied for C programs that use the standard input/output or memory management library functions. Table 4-1 shows the low-level interface routines used by standard library functions.

Table 4-1 Low-Level Interface Routines

No.	Name	Explanation
1	**open**	Opens files
2	**close**	Closes files
3	**read**	Reads data from a file
4	**write**	Writes data to a file
5	**lseek**	Sets the file read/write address for data
6	**sbrk**	Allocates a memory area

Refer to the attached Standard Library Memory Stack Size Listing for details on low-level interface routines required for each C library function.

Initialization of low-level interface routines must be performed when the program is started. For more information, see the explanation concerning the **_INIT_LOWLEVEL** function in section 4.4, Initializing C Library Functions (**_INITLIB**).

The rest of this section explains the basic concept of low-level input and output, and gives the specifications for each interface routine. Refer to appendix E, Examples of Low-Level Interface Routines, for details on the low-level interface routines that run on the SH-series simulator debugger.

Note: The **open**, **close**, **read**, **write**, **lseek**, and **sbrk** are reserved words for low-level interface routines. Do not use these words in C programs.

(1) Concept of I/O Operations

Standard input/output library functions manage files using the **FILE**-type data. Low-level interface routines manage files using file numbers (positive integers) which correspond directly to actual files.

The open routine returns a file number for a given file name. The open routine must determine the following, so that other functions can access information about a file using the file number:

① File device type (console, printer, disk, etc.)
(For a special device such as a console or printer file, the user chooses a specific file name that can be recognized uniquely by the **open** routine.)
② Information such as the size and address of the buffer used for the file
③ For a disk file, the offset (in bytes) from the beginning of the file to the next read/write position.

The start position for read/write operations is determined by the **lseek** routine according to the information determined by the **open** routine.

If buffers are used, the **close** routine outputs the contents to their corresponding files. This allows the areas of memory allocated by the **open** routine to be reused.

(2) Low-Level Interface Routine Specifications

This section explains the specifications for creating low-level interface routines, gives examples of actual interfaces and explains their operations, and notes on implementation.

The interface for each routine is shown using the format below.
Create each interface routine by assuming that the prototype declaration is made.

Example:

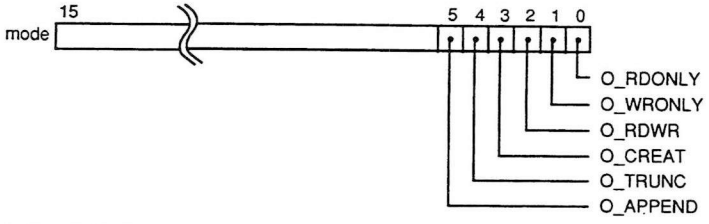
(Routine name)				
Purpose	(Purpose of the routine)			
Interface	(Shows the interface as a C function declaration)			
Parameters	No.	Name	Type	Meaning
	1	(Parameter name)	(Parameter type)	(Meaning of the parameter)
	⋮	⋮	⋮	⋮
Return value	Type	(Type of return value)		
	Normal	(Return value for normal termination)		
	Abnormal	(Return value for abnormal termination)		

(a) **open** routine				
Purpose	Opens a file			
Interface	`int open (char *name,` `int mode);`			
Parameters	No.	Name	Type	Meaning
	1	`name`	Pointer to **char**	String literal indicating a file name
	2	`mode`	**int**	Processing specification
Return value	Type	**int**		
	Normal	File number of the file opened		
	Abnormal	−1		

Explanation:

The **open** routine opens the file specified by the first parameter (file name) and returns a file number. The **open** routine must determine the file device type (console, printer, disk, etc.) and assign this information to the file number. The file type is referenced using the file number each time a read/write operation is performed.

The second parameter (mode) gives processing specifications for the file. The effect of each bit of this parameter is explained below:



① O_RDONLY (bit 0)
If this bit is 1, the file becomes read only.

② O_WRONLY (bit 1)
If this bit is 1, the file becomes write only.

③ O_RDWR (bit 2)
If this bit is 1, the file becomes read/write.

④ O_CREAT (bit 3)
If this bit is 1 and the file indicated by the file name does not exist, a new file is created.

⑤ O_TRUNC (bit 4)
If this bit is 1 and the file indicated by the file name exists, the file contents are discarded and the file size is set to zero.

⑥ O_APPEND (bit 5)
If this bit is 1, the read/write position is set to the end of the file. If this bit is 0, the read/write position is set to the beginning of the file.

An error is assumed if the file processing specifications contradict with the actual characteristics of the file.

The **open** routine returns a file number (positive integer) which can be used by the **read**, **write**, **lseek**, and **close** routines, provided the file opens normally. The relationship between file numbers and actual files must be managed by the low-level interface routines. The **open** routine returns a value of –1 if the file fails to open properly.

(b) **close** routine				
Purpose	Closes a file			
Interface	`int close(int fileno);`			
Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the file to be closed
Return value	Type		**int**	
	Normal		0	
	Abnormal		–1	

Explanation:

The file number, determined by the **open** routine, is given as the parameter.

The area of memory allocated by the **open** routine for file management information is freed, so that it can be reused. If buffers are used, the contents are output to their corresponding files.

Zero is returned if the file closes normally. Otherwise, –1 is returned.

(c) **read** routine				
Purpose	Reads data from a file			
Interface	`int read (int fileno,` `char *buf,` `unsigned int count);`			
Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the file to be read
	2	`buf`	Pointer to **char**	Area to be used to store the read data
	3	`count`	**unsigned int**	Byte length of data to be read
Return value	Type		**int**	
	Normal		Byte length of the data actually read	
	Abnormal		−1	

Explanation:

The **read** routine loads data from the file indicated by the first parameter (**fileno**) into the area indicated by the second parameter (**buf**). The amount of data to be read is indicated by the third parameter (**count**).

If an end of file is encountered during a read, less than the specified number of bytes are read.

The file read/write position is updated using the byte length of the data actually read.

If data is read normally, the routine returns the number of bytes of the data read. Otherwise, the **read** routine returns a value of −1.

(d) **write** routine

Purpose	Writes data to a file			
Interface	`int write (int fileno,` `char *buf,` `unsigned int count);`			
Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number
	2	`buf`	Pointer to **char**	Area storing data to be written in the file
	3	`count`	**unsigned int**	Byte length of the data to be written
Return value	Type		**int**	
	Normal		Byte length of the data actually written	
	Abnormal		−1	

Explanation:

The **write** routine outputs data, whose byte length is indicated by the third parameter (**count**), from the area indicated by the second parameter (**buf**) into the file indicated by the first parameter (**fileno**).

If the device (such as a disk) where a file is stored becomes full, data less than the specified byte length is written to the file. If zero is returned as the byte length of data actually written several times, the routine assumes that the device is full and sends a return value of −1.

The file read/write position must be updated using the byte length of data actually written.

If the routine ends normally, it returns the byte length of data actually written. Otherwise, the routine returns a value of −1.

(e) **lseek** routine

Purpose	Determines the next read/write position in a file			
Interface	`long lseek (int fileno,` `long offset,` `int base);`			
Parameters	No.	Name	Type	Meaning
	1	`fileno`	**int**	File number of the target file
	2	`offset`	**long**	Offset in bytes from specified point in the file
	3	`base`	**int**	Base used for offset (bytes)
Return value	Type		**long**	
	Normal		The offset (bytes) from the beginning of the file for the next read/write position	
	Abnormal		−1	

Explanation:

The **lseek** routine determines the next read/write position as an offset in bytes. The next read/write position is determined according to the third parameter (**base**) as follows:

① Base = 0
The second parameter gives the new offset relative to the beginning of the file.

② Base = 1
The second parameter is added to the current position to give the new offset.

③ Base = 2
The second parameter is added to the file size to give the new offset.

An error occurs if the file is on an interactive device (such as a console or printer), the new offset value is negative, or the new offset value exceeds the file size in the case of ① or ②, above.

If **lseek** correctly determines a new file position, the new offset value is returned. This value indicates the new read/write position relative to the beginning of the file. Otherwise, the **lseek** routine returns a value of −1.

(f) **sbrk** routine				
Purpose	Allocates a memory area			
Interface	`char *sbrk(unsigned long size);`			
Parameters	No.	Name	Type	Meaning
	1	`size`	**unsigned long**	Size of the area to be allocated
Return value	Type		Pointer to **char**	
	Normal		Start address of the allocated area	
	Abnormal		(char *) – 1	

Explanation:

The size of the area to be allocated is given as a parameter.

Create the **sbrk** routine so that consecutive calls allocate consecutive areas beginning with the lowest available address.
An error will occur if there is insufficient memory.

If the routine ends normally, it returns the start address of the allocated area. Otherwise, the routine returns (char *) – 1.

PART IV ERROR MESSAGES

Section 1 Error Messages Output by the C Compiler

The C compiler checks C source programs for errors. This section explains the format and meaning of error messages that may be generated during compile time, and gives appropriate programmer responses.

1.1 Error Message Format

Error messages are output to the standard output file (normally a terminal). Figures 1-1 and 1-2 show the formats used for error messages.

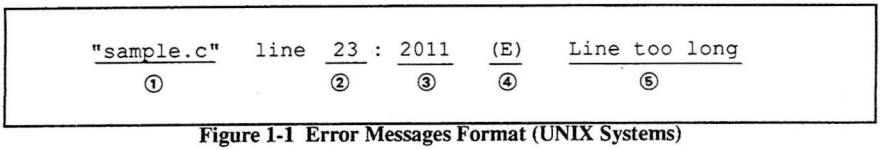


Figure 1-1 Error Messages Format (UNIX Systems)

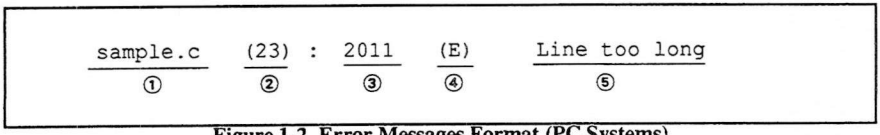


Figure 1-2 Error Messages Format (PC Systems)

Explanation:

① File name
File name (sample.c) of the source program in which the error was detected.

② Line number
Line number (23) where the error was detected.

③ Error number
This number is unique to the error message. See section 1.3, List of Error Messages, for details on the errors and appropriate programmer responses.

④ Message level
The severity of the error. See section 1.2, Message Levels, for details.

⑤ Message text
This describes the error.

Note: When an error not related to the source program has occurred (e.g., an error internal to the compiler), the file name is not output; for the line number here, 0 is output in UNIX systems, and nothing is output in PC systems.

1.2 C Compiler Action and Programmer Response for Each Error Level

Error messages are classified into the following four levels according to their severity. Table 1-1 shows C compiler action for each level of errors.

Table 1-1 C Compiler Action and Programmer Response for Each Error Level

No.	Error Level	Error Meaning	Symbol	Error Number	Object Program Output	Processing Continues	User Response
1	Warning	A mistake with respect to language specifications : The compiler has performed error recovery.	(W)	1000 to 1999	Yes	Yes	Check the list of error messages to decide whether error recovery performed by the C compiler is correct. If necessary, modify and recompile the source program.
2	Error	A mistake in language specifications	(E)	2000 to 2999	No	Yes	Correct the error and recompile the source program.
3	Fatal	The source program exceeds the limit of the C compiler	(F)	3000 to 3999	No	No	Correct the error and recompile the source program.
4	Internal	An error has occurred in an internal process of the C compiler	—	4000 to 4999	No	No	Contact the sales office or representative where the C compiler was purchased.

1.3 List of Error Messages

This section gives lists of error messages in order of error number. A list of error messages are provided for each level of errors.

Example:

Error Number	Message	Explanation
① 2226	② `Scalar required for an "operator"`	③ Binary operator && or || is used in an expression that is not scalar. ④ S: Assumes that the result is **int** and continues processing. ⑤ P: Specify a scalar expression as the operand.

① Error Number

② Error Message

This message is sent to the standard output device (normally a terminal).

③ Explanation

This gives more details about the error.

④ System Action

This indicates the reaction of the C compiler to the error.

⑤ Programmer Response

This indicates to the programmer how to resolve the error.

(1) Warning-Level Messages

Error No.	Message	Explanation
1000	`Illegal pointer assignment`	A pointer is assigned to a pointer with a different data type. S: Sets the left hand side to the internal representation of the right hand side pointer. The resultant type is the same as the data type of the left pointer. P: Use the cast operator to specify explicit type conversion.
1001	`Illegal comparison in "operator"`	The operands of the binary operator == or != are a pointer and an integer other than 0. S: Selects an internal representation for the operands. P: Specify the correct type for the operands.
1002	`Illegal pointer for "operator"`	The operands of the binary operator ==, !=, >, <, >=, or <= are pointers assigned to different types. S: Assumes that the operands are pointers assigned to the same type. P: Use a cast operator so that the same operand type will be used.
1005	`Undefined escape sequence`	An undefined escape sequence (a character following a backslash) is used in a character constant or string literal. S: Ignores the backslash. P: Remove the backslash or specify the correct escape sequence.
1007	`Long character constant`	The length of a character constant is 2 characters. S: Uses the specified characters. P: Check that the correct character constant is specified.

Error No.	Message	Explanation
1008	`Identifier too long`	An identifier's length exceeds 31 characters. S: Uses the first 31 characters and ignores the rest. P: Use identifiers with 31 or less characters.
1010	`Character constant too long`	The length of a character constant exceeds four characters. S: Uses the first four characters and ignores the rest. P: Use character constant with four or less characters.
1012	`Floating point constant overflow`	The value of a floating-point constant exceeds the limit. S: Assumes the internally represented value corresponding to $+\infty$ or $-\infty$ depending on the sign of the result. P: Specify floating-point constants within their limits.
1013	`Integer constant overflow`	The value of **unsigned long** integer constant exceeds the limit. S: Ignores the overflow and uses the remaining bits. P: Specify integer constants within their limits.
1014	`Escape sequence overflow`	The value of an escape sequence indicating a bit pattern in a character constant or string literal exceeds 255. S: Uses the low order byte. P: Change the value of the escape sequence to 255 or lower.

Error No.	Message	Explanation
1015	`Floating point constant underflow`	The absolute value of a floating-point constant is less than the lower limit. S: Assumes 0.0 as the value of the constant. P: Change the value of the constant to 0.0 or specify a constant whose value can be represented.
1016	`Argument mismatch`	The data type assigned to a pointer specified as a formal parameter in a prototype declaration differs from the data type assigned to a pointer used as the corresponding actual parameter in a function call. S: Uses the internal representation of the pointer used for the function call actual parameter. P: Use a cast operator for the function call actual parameter to convert the formal parameter to the type specified in the prototype declaration.
1017	`Return type mismatch`	The function return type and the expression type in a **return** statement are pointers but the data types assigned to these pointers are different. S: Uses the internal representation of the pointer specified in the **return** statement expression. P: Use a cast operator for the expression specified in the **return** statement expression to convert it to the type of the function return value.
1019	`Illegal constant expression`	The operands of the relational operator <, >, <=, or >= in a constant expression are pointers to different data types. S: Assumes 0 as the result value. P: Use an expression other than a constant expression to obtain the correct result.

Error No.	Message	Explanation
1020	`Illegal constant expression of "-"`	The operands of the binary operator – in a constant expression are pointers to different data types. S: Assumes 0 as the result value. P: Use an expression other than a constant expression to obtain the correct result.
1200 :	`Division by floating point zero`	Division by the floating-point number 0.0 is carried out in the evaluation of a constant expression. S: Assumes the internal representation of the value corresponding to $+\infty$ or $-\infty$ depending on the sign of the operands. P: Specify the correct constant expression.
1201	`Ineffective floating point operation`	Invalid floating-point operations such as $\infty - \infty$ or 0.0/0.0 are carried out in a constant expression. S: Assumes the internal representation of not a number to indicate the result of an ineffective operation. P: Correct the constant expression.
1300	`Command parameter specified twice`	The same C compiler option is specified more than once. S: Uses the last specified compiler option. P: Check that options are specified correctly.
1301	`Too many define options`	The number of macro names specified as suboptions in the define option exceeds 16. S: Uses the first 16 suboptions. P: Define the 17th and subsequent macro names using #define directives at the beginning of the source program.

(2) Error-Level Messages

Error No.	Message	Explanation
2000	`Illegal preprocessor keyword`	An illegal keyword is used in a preprocessor directive. S: Ignores the line containing the preprocessor directive. P: Correct the keyword in the preprocessor directive.
2001	`Illegal preprocessor syntax`	There is an error in preprocessor directive or in a macro call specification. S: Ignores the line containing the preprocessor directive or macro call. If there is an error in a constant expression used in the preprocessor directive, the system assumes that the constant expression is 0. P: Specify the correct preprocessor directive or macro call.
2002	`Missing ","`	A comma (,) is not used to delimit two arguments in a #define directive. S: Assumes that there is a comma. P: Insert a comma.
2003	`Missing ")"`	A right parenthesis ")" does not follow a name in a defined expression. The defined expression determines whether the name is defined by a #define directive. S: Assumes that there is a right parenthesis. P: Insert a right parenthesis.
2004	`Missing ">"`	A right angle bracket (>) does not follow a file name in an #include directive. S: Assumes that there is a right angle bracket. P: Insert a right angle bracket.

Error No.	Message	Explanation
2005	`Cannot open include file "file name"`	The file specified by an #include directive cannot be opened. S: Ignores the #include directive. P: Specify the correct file name. If the file name is correct, check that the file does not have write only status.
2006	`Multiple #define's`	The same macro name is redefined by #define directives. S: Ignores the second #define directive. P: Modify one of the macro names or delete one of the #define directives.
2008	`Processor directive #elif mismatches`	There is no #if, #ifdef, #ifndef, or #elif directive corresponding to an #elif directive. S: Ignores the #elif directive. P: Insert the corresponding preprocessor directive or delete the #elif directive.
2009	`Processor directive #else mismatches`	There is no #if, #ifdef, or #ifndef directive corresponding to an #else directive. S: Ignores the #else directive. P: Insert the corresponding preprocessor directive or delete the #else directive.
2010	`Macro parameters mismatch`	The number of macro call arguments is not equal to the number of macro definition arguments. S: Ignores the excess arguments if there are too many, or assumes blank string literals if the number of arguments is insufficient. P: Specify the correct number of macro arguments.

Error No.	Message	Explanation
2011	`Line too long`	After macro expansion, a source program line exceeds the limit of 4095 characters for UNIX systems, and 512 characters for PC systems. S: Ignores the 4096th and subsequent characters. P: Separate the line so that the length of each resulting line is within the limit after macro expansion.
2012	`Keyword as a macro name`	A preprocessor keyword is used as a macro name in a #define or #undef directive. S: Ignores the #define or #undef directive P: Change the macro name.
2013	`Processor directive #endif mismatches`	There is no #if, #ifdef, or #ifndef directive corresponding to an #endif directive. S: Ignores the #endif directive. P: Check that the #endif directive is used correctly.
2014	`Missing #endif`	There is no #endif directive corresponding to an #if, #ifdef, or #ifndef directive, and the end of file is detected. S: Assumes that there is an #endif directive. P: Insert an #endif directive.
2016	`Preprocessor constant expression too complex`	The total number of operators and operands in a constant expression specified by an #if or #elif directive exceeds the limit of 512 for UNIX systems, and 210 for PC systems. S: Assumes the value of the constant expression to be 0. P: Correct the constant expression so that the number of operators and operands is less than or equal to the limit.

Error No.	Message	Explanation
2017	`Missing "`	A closing double quotation mark (") does not follow a file name in an #include directive. S: Assumes that there is a closing double quotation mark P: Insert a closing double quotation mark.
2018	`Illegal #line`	The line count specified by a #line directive exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. S: Ignores the #line directive. P: Modify the program so that the line count is less than or equal to the limit.
2019	`File name too long`	The length of a file name exceeds 128 characters. S: Uses the first 128 characters as the file name. P: Change the file name so that the length is less than or equal to 128 characters.
2020	`System identifier "name" redefined`	The name of the defined symbol is the same as that of the run time routine. S: Continues processing as a unique symbol. P: Define the symbol with a different name from that of the run time routine.
2100	`Multiple storage classes`	Two or more storage class specifiers are used in a declaration. S: Uses the first storage class specifier and ignores others. P: Specify the correct storage class specifier.

Error No.	Message	Explanation
2101	Address of register	The unary operator & is used on a **register** variable. S: Assumes that the **auto** storage class is specified for the variable and continues processing. P: Modify the declaration so that the storage class of the variable is **auto** .
2102	Illegal type combination	A combination of type specifiers is illegal. S: Uses the first and longest legal combination of type specifiers and ignores the rest. P: Change the type specifiers to a legal combination.
2103	Bad self reference structure	A **struct** or **union** member has the same data type as its parent. S: Assumes the data type of the member is **int**. P: Declare the correct data type for the member.
2104	Illegal bit field width	A constant expression indicating the width of a bit field is not an integer or it is negative. S: Ignores the bit field width specification and assumes that the member is not a bit field. P: Specify the correct width for the bit field.
2105	Incomplete tag used in declaration	An incomplete tag name declared with a **struct** or **union**, or an undeclared tag name is used in a **typedef** declaration or in the declaration of a data type not assigned to a pointer or to a function return value. S: Assumes that the incomplete or undeclared tag name is an **int**. P: Declare the incomplete or undeclared tag name.

Error No.	Message	Explanation
2106	`Extern variable initialized`	A compound statement specifies an initial value for an **extern** storage class variable. S: Ignores the initial value. P: Specify the initial value for the external definition of the variable.
2107	`Array of function`	An array with a function member type is specified. S: Ignores the function or array type. P: Specify the correct type.
2108	`Function returning array`	A function with an array return value type is specified. S: Ignores the function or array type. P: Specify the correct type.
2109	`Illegal function declaration`	A storage class other than **extern** is specified in the declaration of a function variable used in a compound statement. S: Assumes **extern** as the storage class. P: Specify the correct storage class.
2110	`Illegal storage class`	The storage class in an external definition is specified as **auto** or **register**. S: Assumes that the storage class is **extern**. P: Specify the correct storage class.
2111	`Function as a member`	A member of a **struct** or **union** is declared as a function. S: Assumes **int** as the member type. P: Declare the correct member type.

Error No.	Message	Explanation
2112	`Illegal bit field`	The type specifier for a bit field is illegal. **char**, **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, or a combination of **const** or **volatile** with one of the above types is allowed as a type specifier for a bit field. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Specify the correct type.
2113	`Bit field too wide`	The width of a bit field is greater than the size (8, 16, or 32 bits) indicated by its type specifier. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Specify the correct bit field width.
2114	`Multiple variable declarations`	A variable name is declared more than once in the same scope. S: Uses the first declaration and ignores subsequent declarations. P: Keep one of the declarations and delete or modify the rest.
2115	`Multiple tag declarations`	A **struct**, **union**, or **enum** tag name is declared more than once in the same scope. S: Uses the first declaration and ignores subsequent declarations. P: Keep one of the tag name declarations and delete or modify the rest.
2117	`Empty source program`	There are no external definitions in the source program. S: Terminates processing. P: Specify and compile the correct source program.

Error No.	Message	Explanation
2118	`Prototype mismatch`	A function type differs from the one specified in the declaration. S: Ignores the current declaration if the function prototype declaration is being processed. Ignores the previous declaration if the declaration of an external function definition is being processed. P: Correct the declaration so that the function types match.
2119	`Not a parameter name`	An identifier not in the function parameter list is declared as a parameter. S: Ignores the parameter declaration. P: Check that the function parameter list matches all parameter declarations.
2120	`Illegal parameter storage class`	A storage class other than **register** is specified in a function parameter declaration. S: Ignores the storage class specifier. P: Delete the storage class specifier.
2121	`Illegal tag name`	The combination of a tag name and **struct**, **union**, or **enum** differs from the declared combination. S: Assumes **struct**, **union**, or **enum** depending on the tag name type. P: Specify the correct combination of a tag name and a **struct**, **union**, or **enum**.
2122	`Bit field with 0`	The width of a bit field which is a member of a **struct** or **union** is 0. S: Ignores the bit field specification and assumes that the member is not a bit field. P: Delete the member name or specify the correct bit field width.

Error No.	Message	Explanation
2123	`Undefined tag name`	An undefined tag name is specified in an **enum** declaration. S: Ignores the declaration. P: Specify the correct tag name.
2124	`Illegal enum value`	A non-integral constant expression is specified as a value for an **enum** member. S: Ignores the value specification. P: Change the expression to an integer constant expression.
2125	`Function returning function`	A function with a function return value is specified. S: Ignores one of the function types. P: Specify the correct type.
2126	`Illegal array size`	The value that specifies the number of elements in an array is other than an integer between 1 and 2147483647. S: Assumes the number of array elements to be one. P: Specify a valid number of array elements.
2127	`Missing array size`	The number of elements in an array is not specified where it is required. S: Assumes that the number of array element is one. P: Specify the number of array elements.

Error No.	Message	Explanation
2128	`Illegal pointer declaration for "*"`	A type specifier other than **const** or **volatile** is specified following an asterisk (*), which indicates a pointer declaration. S: Ignores the type specifier following the asterisk. P: Specify the correct type specifier following the asterisk.
2129	`Illegal initializer type`	The initial value specified for a variable is not a type that can be assigned to the variable. S: Does not initialize the variable. P: Specify the correct type of initial value.
2130	`Initializer should be constant`	A value other than a constant expression is specified as either the initial value of a **struct**, **union**, or array variable or as the initial value of a static variable. S: Does not initialize the variable. P: Specify a constant expression as the initial value.
2131	`No type nor storage class`	Storage class and type specifiers are not given in an external data definition. S: Assumes **int** as the type specifier. P: Insert the storage class or type specifier.
2132	`No parameter name`	A parameter is declared even though the function parameter list is empty. S: Ignores the parameter declaration. P: Insert the parameter name in the function parameter list or delete the parameter declaration.

Error No.	Message	Explanation
2133	`Multiple parameter declarations`	Either a parameter name is declared in a function definition parameter list more than once or a parameter is declared inside and outside the function declarator. S: Uses the first declaration if a parameter is declared more than once in the function parameter list. Uses the declaration inside the function declarator if a parameter is declared inside and outside the function declarator. P: Keep one of the declarations and delete the rest.
2134	`Initializer for parameter`	An initial value is specified in the declaration of an parameter. S: Does not use the initial value specification. P: Delete the initial value specification.
2135	`Multiple initialization`	A variable is initialized more than once. S: Ignores the second and subsequent initialization directives. P: Delete any redundant directives.
2136	`Type mismatch`	An **extern** or **static** variable or function is declared more than once with different data types. S: Uses the type specified in the definition declaration where a definition is declared. Otherwise, the data type specified in the first declaration is used. P: Use the same data type in the declarations.

Error No.	Message	Explanation
2137	`Null declaration for parameter`	An identifier is not specified in the function parameter declaration. S: Ignores the corresponding parameter declaration. P: Delete the parameter declaration or insert the correct parameter name.
2138	`Too many initializers`	The number of initial values specified for a **struct** or array is greater than the number of **struct** members or array elements. This error also occurs if two or more initial values are specified when the first members of a **union** are scalar. S: Uses only the initial values corresponding to the number of **struct** members, array elements, or the first members of **union**. The rest are ignored. P: Specify the correct number of initial values.
2139	`No parameter type`	A type is not specified in a function parameter declaration. S: Assumes **int** as the parameter declaration type. P: Specify the correct type for the parameter declaration.
2140	`Illegal bit field`	A bit field is used in a **union**. S: Ignores the bit field. P: Use the bit field in a **struct**.
2141	`Illegal bit field`	An unnamed bit field is used as the first member of a **struct**. S: Ignores the bit field. P: Specify the name of the bit field.

Error No.	Message	Explanation
2142	`Illegal void type`	**void** is used illegally. S: Assumes that **void** is **int**. P: **void** can only be used in the following cases: (1) To specify a type assigned to a pointer (2) To specify a function return value type (3) To explicitly specify that a function whose prototype is declared does not have a parameter
2143	`Illegal static function`	A **static** storage class function has no definition in the source program. S: Ignores the function declaration. P: Either delete the function declaration or define the function.
2144	`Type mismatch`	**extern** variables or functions with the same names are declared with different data types in different valid ranges. S: The currently declared variable or function type is valid within the range that can be referenced. However, when linked with another file, the valid data type is determined as shown below. (1) If there is a declaration that acts as a definition, that data type is valid. (2) If there is no declaration that acts as a definition: — The previously declared data type is valid when the current declaration is in the function. — The currently declared data type is valid when the current declaration is not in the function. P: Declare the same data types for **extern** variables or functions.

Error No.	Message	Explanation
2200	`Index not integer`	An array index expression type is not an integer. S: Assumes that the type is **int**. P: Specify an integer expression for the array index.
2201 :	`Cannot convert parameter`	The nth parameter of a function call cannot be converted to the type of parameter specified in the prototype declaration. S: Assumes that the correct parameter type is specified and continues processing. P: Specify an expression whose type corresponds to the one specified in the prototype declaration.
2202	`Number of parameters mismatch`	The number of parameters for a function call is not equal to the number of parameters specified in the prototype declaration. S: Assumes that the number of parameters for the function call is equal to the number of parameters specified in the prototype declaration, and continues processing. P: Specify the correct number of parameters.
2203	`Illegal member reference for "."`	The expression to the left of the (.) operator is not a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Use a **struct** or **union** expression to the left of the (.) operator.

Error No.	Message	Explanation
2204	`Illegal member reference for "->"`	The expression to the left of the –> operator is not a pointer to a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Use an expression which deals with pointer to **struct** or **union** to the left of the –> operator according to the member.
2205	`Undefined member name`	An undeclared member name is used to reference a **struct** or **union**. S: Assumes that the member is not referenced and continues processing. P: Specify the correct member name.
2206	`Modifiable lvalue required for "operator"`	The operand for a unary prefix or suffix operator ++ or – – has a left value that cannot be assigned (a left value whose type is not array or **const**). S: Assumes that the expression with a left value that can be assigned is specified as an operand and continues processing. P: Specify an expression, whose left value can be assigned, as an operand.
2207	`Scalar required for "!"`	The unary operator ! is used on an expression that is not scalar. S: Assumes **int** as the type of the result and continues processing. P: Use a scalar expression as the operand.
2208	`Pointer required for "*"`	The operand for the unary operator * is an expression of pointer to **void** or is not an expression of pointer. S: Ignores *. P: Use an operand that is an expression other than pointer to **void**.

Error No.	Message	Explanation
2209	`Arithmetic type required for "operator"`	The unary operator + or – is used on a non-arithmetic expression. S: Assumes that the operand type is **int** and continues processing P: Change the expression to an arithmetic expression
2210	`Integer required for "~"`	The unary operator ~ is used on a non-integral expression. S: Assumes that the result type is **int** and continues processing. P: Change the expression to an integral expression.
2211	`Illegal sizeof`	A **sizeof** operator is used for a bit field member, function, **void**, or array with an undefined size. S: Assumes **int** as the operand type and continues processing. P: A **sizeof** operator cannot be used to obtain the size of a bit field, function, **void**, or array with an undefined size. Use an appropriate operand.
2212	`Illegal cast`	Either array, **struct**, or **union** is specified in a cast operator, or the operand of a cast operator is **void**, **struct**, or **union** and cannot be converted. S: Assumes that the result is **int** and continues processing. P: Cast operation can only be performed on scalar data items. Use appropriate operands.

Error No.	Message	Explanation
2213	`Arithmetic type required for "operator"`	The binary operator *, /, *=, or /= is used in an expression that is not arithmetic. S: Assumes **int** as the result and continues processing. P: Specify arithmetic expressions as the operands.
2214	`Integer required for "operator"`	The binary operator <<, >>, &, |, ^, %, <<=, >>=, &=, | =, ^=, or %= is used in an expression that is not an integer expression. S: Assumes **int** as the result type and continues processing. P: Specify integer expressions as the operands.
2215	`Illegal type for "+"`	The combination of operand types used with the binary operator + is illegal. S: Assumes the result type is **int** and continues processing. P: Specify a correct type of operands. Only the following type combinations are allowed for the binary operator +: — Two arithmetic operands — Pointer and integer
2216	`Illegal type for parameter`	**void** is specified for a function call parameter type. S: Ignores the parameter type and continues processing. P: Specify a function call parameter type so that a value can be passed to the function.

Error No.	Message	Explanation
2217	`Illegal type for "-"`	The combination of operand types used with the binary operator – is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following three combinations are allowed for the binary operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) The first operand is a pointer and the second operand is an integer.
2218	`Scalar required`	The first operand of the conditional operator ?: is not a scalar. S: Assumes that the first operand is a scalar and continues processing. P: Specify a scalar expression as the first operand.

Error No.	Message	Explanation
2219	`Type not compatible in "?:"`	The types of the second and third operands of the conditional operator ?: do not match with each other. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only one of the following six combinations is allowed for the second and third operands when using the ?: operator: (1) Two arithmetic operands (2) Two **void** operands (3) Two pointers assigned to the same data type (4) A pointer and an integer constant whose value is 0 or another pointer that is assigned to **void** that was converted from an integer constant whose value is 0 (5) A pointer and another pointer assigned to **void** (6) Two **struct** or **union** variables with the same data type
2220	`Modifiable lvalue required for "operator"`	An expression whose left value cannot be assigned (a left value whose type is not array or **const**) is used as an operand of an assignment operator =, *=, /=, %=, +=, –=, <<=, >>=, &=, ^=, or | =. S: Assumes that the left expression whose left value can be assigned is used and continues processing. P: Specify a left expression whose left value can be assigned.

Error No.	Message	Explanation
2221	`Illegal type for "operator"`	The operand of the unary suffix operator ++ or -- is function type, a pointer assigned to **void**, or not scalar type. S: Assumes that the result type is **int** and continues processing. P: Use a scalar type that is not a function or a pointer assigned to **void** as the operand.
2222	`Type not compatible for "="`	The operand types for the assignment operator = do not match. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following five type combinations are allowed for the operands of the = assignment operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) The left operand is a pointer and the right operand is an integer constant whose value is 0 or another pointer that is assigned to **void** that was converted from an integer constant whose value is 0. (4) A pointer and another pointer assigned to **void** (5) Two **struct** or **union** variables with the same data type
2223	`Incomplete tag used in expression`	An incomplete tag name is used for a **struct** or **union** in an expression. S: Assumes that the incomplete tag name is **int** and continues processing. P: Declare the tag name.

Error No.	Message	Explanation
2224	`Illegal type for assign`	The operand types of the assignment operator += or –= are illegal. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following two type combinations are allowed as operands for the assignment operator += or –=: (1) Two arithmetic operands (2) The left operand is a pointer and the right operand is an integer.
2225	`Undeclared name`	An undeclared name is used in an expression. S: Assumes that the name is declared as an **int** external identifier and continues processing. P: Either declare the name or modify it so that it corresponds with one of the declared names.
2226	`Scalar required for "operator"`	The binary operator && or || is used in a non-scalar expression. S: Assumes that the result type is **int** and continues processing. P: Use scalar expressions as operands.

Error No.	Message	Explanation
2227	`Illegal type for equality`	The combination of operand types for the equality operator == or != is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following three combinations of operand types for the equality operator == or != are allowed: (1) Two arithmetic operands (2) Two pointers assigned to the same data type (3) A pointer and an integer constant whose value is 0 or another pointer assigned to **void**
2228	`Illegal type for comparison`	The combination of operand types for the relational operator >, <, >=, or <= is not allowed. S: Assumes that the result type is **int** and continues processing. P: Specify a correct type combination of operands. Only the following two combinations of operand types are allowed for a relational operator: (1) Two arithmetic operands (2) Two pointers assigned to the same data type
2230	`Illegal function call`	An expression which is not a function type or a pointer assigned to a function type is used for a function call. S: Ignores the actual argument list and the parentheses which indicate this list. P: Specify a function type expression or pointer assigned to a function type correctly.

Error No.	Message	Explanation
2231	`Address of bit field`	The unary operator & is used on a bit field. S: Ignores the bit field, assumes that the unary operator & is correctly specified, and continues processing. P: Correct the expression. A bit field address cannot be used.
2232	`Illegal type for "operator"`	A type that is not a scalar, or that is a pointer assigned to a function or **void** is specified as the operand for the prefix operator ++ or – –. S: Assumes **int** as the result type and continues processing. P: Use an operand that is a scalar other than a pointer assigned to a function or **void**.
2233	`Illegal array reference`	An expression used as an array is not one of the following types: — Array — Pointer assigned to a data type other than a function or **void** S: Ignores the square brackets ([ ]) and the array subscript enclosed. P: When an array subscript is required, specify the correct expression.
2234	`Illegal typedef name reference`	A **typedef** name is used as a variable in an expression. S: Ignores the expression. P: Use **typedef** correctly.
2235	`Illegal cast`	An attempt is made to cast a pointer with a floating-point type. S: Ignores the attempt. P: Cast the pointer with an integer type, then with a floating-point type.

Error No.	Message	Explanation
2236	`Illegal cast in constant`	An attempt is made to cast a pointer with a **char** or **short**. S: Ignores the cast operation. P: Use an expression other than a constant one.
2237	`Illegal constant expression`	In a constant expression, a pointer constant is cast with an integer and the result is manipulated. S: Assumes that the conversion is not specified and continues processing. P: Use an expression other than a constant expression.
2238	`Lvalue or function type required for "&"`	The unary operator & is used on the left value or is used in an expression other than function type. S: Assumes that an expression with a left value is specified as the operand and continues processing. P: Specify an expression that has a left value or a function type expression as the operand.
2300	`Case not in switch`	A **case** label is specified outside a **switch** statement. S: Ignores the **case** label. P: Specify the **case** label in a **switch** statement.
2301	`Default not in switch`	A **default** label is specified outside a **switch** statement. S: Ignores the **default** label. P: Specify the **default** label in a **switch** statement.

Error No.	Message	Explanation
2302	`Multiple labels`	A label is defined more than once in a function. S: Ignores redundant label definitions. P: Keep one label name and delete or modify the other.
2303	`Illegal continue`	A **continue** statement is specified outside a **while**, **for**, or **do** statement. S: Ignores the **continue** statement. P: Only use the **continue** statement in a **while**, **for**, or **do** statement.
2304	`Illegal break`	A **break** statement is specified outside a **while**, **for**, **do**, or **switch** statement. S: Ignores the **break** statement. P: Only use the **break** statement in a **while**, **for**, **do**, or **switch** statement.
2305	`Void function returns value`	A **return** statement specifies a return value for a function with a **void** return type. S: Ignores the **return** statement expression. P: For a function with a **void** return type, do not specify an expression in a **return** statement or do not use the **return** statement.
2306	`Case label not constant`	A **case** label expression is not an integer constant expression. S: Ignores the **case** label. P: Use an integer constant expression for the **case** label.
2307	`Multiple case labels`	Two or more **case** labels with the same value are used in one **switch** statement. S: Ignores redundant **case** labels. P: Modify the **switch** statement so that each **case** label has a unique value.

Error No.	Message	Explanation
2308	`Multiple default labels`	Two or more **default** labels are specified for one **switch** statement. S: Ignores redundant **default** labels. P: Modify the **switch** statement so that it has only one **default** label.
2309	`No label for goto`	There is no label corresponding to the destination specified by a **goto** statement. S: Continues processing. P: Specify the correct label in the **goto** statement.
2310	`Scalar required`	The control expression (that determines statement execution) for a **while**, **for**, or **do** statement is not a scalar. S: Assumes that an **int** control expression is specified and continues processing. P: Use a scalar expression as the control expression for a **while**, **for**, or **do** statement.
2311	`Integer required`	The control expression (that determines statement execution) for a **switch** statement is not an integer. S: Assumes that an **int** control expression is specified and continues processing. P: Use an integer expression as the control expression for the **switch** statement.
2312	`Missing (`	The control expression (that determines statement execution) does not follow a left parenthesis "(" for an **if**, **while**, **for**, **do**, or **switch** statement. S: Assumes that the control expression follows a left parenthesis "(" and continues processing. P: Specify the control expression for an **if**, **while**, **for**, **do**, or **switch** statement and enclose it in parentheses.

Error No.	Message	Explanation
2313	`Missing ;`	A **do** statement is ended without a semicolon (;). S: Assumes that the **do** statement ends with a semicolon (;) and continues processing. P: Place a semicolon (;) at the end of the **do** statement.
2314	`Scalar required`	A control expression (that determines statement execution) for an **if** statement is not a scalar. S: Assumes that an **int** control expression is specified and continues processing. P: Use a scalar expression as the control expression for **if** statement.
2316	`Illegal type for return value`	An expression in a **return** statement cannot be converted to the type of value expected to be returned by the function. S: Assumes that the expression in the **return** statement is the type expected to be returned by the function and continues processing. P: Convert the expression in the **return** statement so that it matches the type of value expected.
2400	`Illegal character "character"`	An illegal character is detected. S: Assumes that the character is a blank character and continues processing. P: Delete the illegal character.

Error No.	Message	Explanation
2401	Incomplete character constant	An end of line indicator is detected in the middle of a character constant. S: Assumes that a quotation mark (') is placed before the end of line indicator and continues processing. P: Correct the character constant.
2402	Incomplete string	An end of line indicator is detected in the middle of a string literal. S: Assumes that a double quotation mark (") is placed before the end of line indicator and continues processing. P: Correct the string literal.
2403	EOF in commment	An end of file indicator is detected in the middle of a comment. S: Assumes that the program ends when the end of file indicator is reached and continues processing. P: End the comment with */.
2404	Illegal character code "character code"	An illegal character code is detected. S: Assumes that the character code is a blank character and continues processing. P: Delete the illegal character code.
2405	Null character constant	There are no characters in a character constant (i.e., no characters are specified between two quotation marks). S: Assumes that "\0" is specified and continues processing. P: Correct the character constant.

Error No.	Message	Explanation
2406	Out of float	The number of significant digits in a floating-point constant exceeds 17. S: Depending on the sign, the system assumes +∞ or −∞. P: Ensure that the number of significant digits in a floating-point constant is less than or equal to 17.
2407	Incomplete logical line	A backslash (\) or a backslash followed by an end of line indicator (\RET) is specified as the last character in a non-empty source file. S: Ignores the last logical line. P: Delete the backslash or continue the physical line.
2500	Illegal token	An illegal token sequence is used. S: Ignores data up to a semicolon (;), left brace ({), right brace (}), comma (,), or keyword (**if**, **while**, **for**, **switch**, **do**, **case**, **default**, **return**, **break**, or **continue**). P: Correct the token sequence.
2501	Division by zero	An integer is divided by zero in a constant expression. S: Assumes a result value of zero and continues processing. P: Modify the constant expression so that an integer is not divided by zero.
2600	character string	An error message specified by string literal #error is output to the list file if nolist option is not specified. S: Continues processing.

Error No.	Message	Explanation
2650	`Invalid pointer reference`	The specified address does not match the required byte alignment. S: Uses the address with the lowest bit masked when accessing word data, and the address with the lowest two bits masked when accessing long word data. P: Specify the address so as to match the byte alignment.
2700	`Function "function name" in #pragma interrupt already declared`	A function already declared as a normal function has been specified with the interrupt function declaration #pragma interrupt. S: Ignores the interrupt function declaration. P: Declare the function as an interrupt function before it is declared as a normal function.
2701	`Multiple interrupt for one function`	A function has been declared as an interrupt function with #pragma interrupt more than once. S: Ignores the interrupt function declaration. P: Delete the declarations following the first one.
2702	`Multiple #pragma interrupt options`	The same type of interrupt specifications have been specified more than once. S: Ignore the interrupt function declaration. P: Delete one of the interrupt specifications.
2703	`Illegal #pragma interrupt declaration`	The specifications for the interrupt function declaration #pragma interrupt are not correct. S: Ignores the interrupt function declaration. P: Specify correctly.

Error No.	Message	Explanation
2704	Illegal reference to interrupt function	An interrupt function is illegally referenced. S: Ignores the attempt to reference the interrupt function. P: An interrupt function cannot normally be referenced. Define another function for referencing.
2705	Illegal parameter in interrupt function	There are different parameter types in an interrupt function. S: Ignores the interrupt function declaration. P: Specify correct parameter types.
2706	Missing parameter declaration in interrupt function	The variables used in the option specification by the interrupt function are not specified. S: Ignores the interrupt function declaration. P: Declare the variables before declaring the interrupt function declaration #pragma interrupt.
2707	Parameter out of range in interrupt function	Parameter tn in an interrupt function exceeds 256. S: Ignores the value of parameter tn. P: Modify the value of parameter tn so it does not exceed 256.
2800	Illegal parameter number in in-line function	The number of parameters used in an intrinsic function does not match the required number. S: Ignores the intrinsic function. P: Specify the correct number of parameters.
2801	Illegal parameter type in in-line function	There are different parameter types in an intrinsic function. S: Ignores the intrinsic function. P: Specify the correct parameter types.

Error No.	Message	Explanation
2802	`Parameter out of range in in-line function`	A parameter exceeds the range that can be specified by an intrinsic function. S: Ignores the intrinsic function. P: Check the range that can be specified for the parameter and specify it correctly.
2803	`Invalid offset value in in-line function`	A parameter is specified improperly by an intrinsic function. S: Ignores the intrinsic function. P: Check the intrinsic function specifications and specify it correctly.

(3) Fatal-Level Messages

Error No.	Message	Explanation
3000	`Statement nest too deep`	The nesting level of an **if**, **while**, **for**, **do**, and **switch** statements exceeds the limit of 32 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3001	`Block nest too deep`	The nesting level of compound statements exceeds the limit of 32 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3002	`#if nest too deep`	The conditional compilation (#if, #ifdef, #ifndef, #elif, and #else) nesting level exceeds the limit of 32 for UNIX systems, and 6 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level is less than or equal to the limit.
3003	`Too many external identifiers`	The number of external identifiers exceeds the limit of 4096 for UNIX systems, and 511 for PC systems. S: Terminates processing. P: Divide the program so that the number of external identifiers is less than or equal to the limit.

Error No.	Message	Explanation
3004	`Too many local identifiers`	The number of effective identifiers (internal identifiers) in one function exceeds the limit of 4096 for UNIX systems, and 512 for PC systems. S: Terminates processing. P: Divide the compound statements so that the number of identifiers declared in one compound statement is less than or equal to the limit.
3005	`Too many macro identifiers`	The number of macro names defined in a #define directive exceeds the limit of 4096 for UNIX systems, and 1024 for PC systems. S: Terminates processing. P: Divide the program so that the number of macro names is less than or equal to the limit.
3006	`Too many parameters`	The number of parameters in either a function declaration or a function call exceeds the limit of 63 for UNIX systems, and 31 for PC systems. S: Terminates processing. P: Divide the compound statements so that the number of identifiers declared in one compound statement is less than or equal to the limit.
3007	`Too many macro parameters`	The number of parameters in a macro definition or a macro call exceeds the limit of 64 for UNIX systems, and 31 for PC systems. S: Terminates processing. P: Modify the program so that the number of macro parameters is less than or equal to the limit.

Error No.	Message	Explanation
3008	`Line too long`	After a macro expansion, the length of a line exceeds the limit of 4095 characters for UNIX systems, and 512 characters for PC systems. S: Terminates processing. P: Divide the line so that its length does not exceed the limit after macro expansion.
3009	`String literal too long`	The length of string literals exceeds 512 characters. The length of string literals is the byte number generated after the specified string is connected continuously. The length of string literals in the source program is not the length of the source program, in the string data. This byte number is located in the string literal data with the expansion sign counted as one character. S: Terminates processing. P: Modify the program so that the total length of string literals does not exceeds 512 bytes.
3010	`Processor directive #include nest too deep`	The nesting level of the #include directive exceeds the limit of 8 for UNIX systems, and 5 for PC systems. S: Terminates processing. P: Ensure that the file inclusion nesting level does not exceed the limit.
3011	`Macro expansion nest too deep`	The nesting level of macro expansion performed by a #define directive exceeds the limit of 32 for UNIX systems, and 16 for PC systems. S: Terminates processing. P: Modify the program so that the nesting level of macro expansion never exceeds the limit. Note that a macro may be defined recursively.

Error No.	Message	Explanation
3012	`Too many function definitions`	The number of function definitions exceeds the limit of 512 for UNIX systems, and 256 for PC systems. S: Terminates processing. P: Divide the program so that the number of function definitions is less than or equal to the limit in one compile unit.
3013	`Too many switches`	The number of **switch** statements exceeds the limit of 256 for UNIX systems, and 128 for PC systems. S: Terminates processing. P: Divide the program so that the number of **switch** statements is less than or equal to the limit in one compile unit.
3014	`For nest too deep`	The nesting level of **for** statements exceeds the limit of 16 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Ensure that the **for** nesting level does not exceed the limit.
3015	`Symbol table overflow`	The number of symbols to be generated by the C compiler exceeds the limit of 8192 for UNIX systems, and 1024 for PC systems. S: Terminates processing. P: Divide the file so that the number of symbols does not exceed the limit.
3016	`Internal label overflow`	The number of internal labels to be generated by the C compiler exceeds the limit of 16384 for UNIX systems, and 2048 for PC systems. S: Terminates processing. P: Divide the file so that the number of internal labels does not exceed the limit.

Error No.	Message	Explanation
3017	`Too many case labels`	The number of **case** labels in one **switch** statement exceeds the limit of 511 for UNIX systems, and 255 for PC systems. S: Terminates processing. P: Ensure that the number of **case** labels does not exceed the limit.
3018	`Too many goto labels`	The number of **goto** labels defined in one function exceeds the limit of 511 for UNIX systems, and 256 for PC systems. S: Terminates processing. P: Ensure that the number of **goto** labels defined in a function does not exceed the limit.
3019	`Cannot open source file "file name"`	A source file cannot be opened. S: Terminates processing. P: Specify the correct file name.
3020	`Source file input error "file name"`	A source or include file cannot be read. S: Terminates processing. P: Check that the file is not read protected.
3021	`Memory overflow`	The C compiler cannot allocate sufficient memory to compile the program. S: Terminates processing. P: Divide the file so that less memory is needed for compilation.
3022	`Switch nest too deep`	The nesting level of **switch** statements exceeds the limit of 16 for UNIX systems, and 15 for PC systems. S: Terminates processing. P: Ensure that the **switch** nesting level does not exceed the limit.

Error No.	Message	Explanation
3023	`Type nest too deep`	The number of types (pointer, array, and function) that qualify the basic type exceeds 16. S: Terminates processing. P: Ensure that the number of types is less than or equal to 16.
3024	`Array dimension too deep`	An array has more than six dimensions. S: Terminates processing. P: Ensure that arrays have no more than six dimensions.
3025	`Source file not found`	A source file name is not specified in the command line. S: Terminates processing. P: Specify a source file name.
3026	`Expression too complex`	An expression is too complex. S: Terminates processing. P: Divide the expression into smaller units.
3027	`Source file too complex`	The nesting level of statements in the program is too deep or an expression is too complex. S: Terminates processing. P: Reduce the nesting level of statements or divide the expression.
3028	`Source line number overflow`	The last source line number exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. S: Terminates processing. P: Modify both the line count specified in the #line directive and the source program so that the last source line number is less than or equal to the limit.

Error No.	Message	Explanation
3029	`Physical line overflow`	The number of physical lines (including the include files) exceeds the limit of 32767 for UNIX systems, and 16383 for PC systems. S: Terminates processing. P: Divide the file so that the number of physical lines does not exceed the limit.
3031	`Data size overflow`	The size of an array or a structure exceeds 2147483647. S: Terminates processing. P: Reduce the size of the array or the structure until it is less than or equal to 2147483647.
3033	`Symbol table overflow`	The number of symbols used for debug information exceeds 30719. S: Terminates processing. P: Divide the file so that the number of symbols does not exceed 30719.
3201	`Object size overflow`	The size of the object program exceeds 4 Gbytes. S: Terminates processing. P: Divide the program so that the size of the object program does not exceed 4 Gbytes.

Error No.	Message	Explanation
3300	Cannot open internal file	An error has occurred in either one of the following cases: (1) An intermediate file internally generated by the C compiler cannot be opened. (2) A file having the same name as the intermediate file already exists. (3) The path name for listing file specifications exceeds 128 characters. (4) A file used internally by the C compiler cannot be opened. S: Terminates processing. P: (1) Check that the intermediate file generated by the C compiler is not being used. (2) Do not use the intermediate file name for other files. (3) Ensure that the path name for listing file specifications does not exceed 128 characters. (4) Check that the disk has sufficient capacity for files.
3301	Cannot close internal file	An intermediate file internally generated by the C compiler cannot be closed. S: Terminates processing. P: (1) Check that there are no mistakes in the compiler installation procedure. (2) Check that there are no abnormalities on the hard disk.
3302	Cannot input internal file	An intermediate file internally generated by the C compiler cannot be read. S: Terminates processing. P: (1) Check that there are no mistakes in the compiler installation procedure. (2) Check that there are no abnormalities on the hard disk.

Error No.	Message	Explanation
3303	Cannot output internal file	An intermediate file internally generated by the C compiler cannot be written. S: Terminates processing. P: Increase the disk size.
3304	Cannot delete internal file	An intermediate file internally generated by the C compiler cannot be deleted. S: Terminates processing. P: Check that the intermediate file generated by the C compiler is not being used.
3305	Invalid command parameter "option name"	An invalid compiler option is specified. S: Terminates processing. P: Specify the correct option.
3306	Interrupt in compilation	An interrupt generated by a CNTL C command (from a standard input terminal) is detected during compilation. S: Terminates processing. P: Input the compile command again.
3307	Compiler version mismatch	File versions in the C compiler do not match. S: Terminates processing. P: Refer to the Install Guide for the installation procedure, and reinstall the C compiler.
3320	Command parameter buffer overflow	The command line specification exceeds 256 characters. S: Terminates processing. P: Ensure that the command line does not exceed 256 characters.

Error No.	Message	Explanation
3321	`Illegal environment variable`	An error has occurred in either of the following cases: (1) The environment variable SHC_LIB is not specified. (2) The file name does not satisfy file name specification rules or the path name exceeds 118 characters. S: Terminates processing. P: (1) Specify the environment variable SHC_LIB. (2) Specify the file name according to file name specification rules. (3) Ensure that the path name does not exceed 118 characters.
4000 to 4999	`Internal error`	An internal error occurs during compilation. S: Terminates processing. P: Report the error occurrence to your local Hitachi dealer.

Section 2 Error Messages Output for the C Library Functions

Some library functions set error numbers to macro **errno** defined by the header file **<stddef.h>** in the C library function when an error occurs during the library function execution. Error messages corresponding to error numbers have already been defined and can be output. The following shows an example of a program which causes an error message output.

Example:

```
#include     <stdio.h>
#include     <string.h>
#include     <stdlib.h>

main ()
{
     FILE *fp

     fp=fopen("file","w");
     fp=NULL;
     fclose(fp);                          /* error occurred */----------①
     printf("%s\n",strerror(errno)) ; /*print error message */--------②
}
```

Description:

1. An error occurs because the file pointer value **NULL** is passed to the **fclose** function as an actual argument. In this case, an error number is set in **errno**.

2. If the error number is passed to the **strerror** function as an actual argument, a pointer to the corresponding error message is returned. Specifying the character string to be output in the **printf** function outputs the error message.

C Library Function Error Messages

Error No.	Message	Explanation	Functions to Set Error Numbers
1100	Data out of range	An overflow occurs.	**atan, cos, sin, tan, cosh, sinh, tanh, exp, fabs, frexp, ldexp, modf, ceil, floor, strtol, atoi, fscanf, scanf, sscanf, atol**
1101	Data out of domain	Results for mathematical parameters are not defined.	**acos, asin, atan2, log, log10, sqrt, fmod, pow**
1102	Division by zero	Division by zero was performed.	**divbs, divws, divls, divbu, divwu, divlu**
1104	Too long string	The length of the character string exceeds 512 characters.	**strtol, strtod, atoi, atol, atof**
1106	Invalid file pointer	**NULL** pointer constant is specified as file pointer value.	**fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror**
1200	Invalid radix	An invalid radix was specified.	**strtol, atoi, atol**
1202	Number too long	The specified number exceeds 17 digits.	**strtod, fscanf, scanf, sscanf, atof**
1204	Exponent too large	The specified exponent exceeds three digits.	**strtod, fscanf, scanf, sscanf, atof**
1206	Normalized exponent too large	The exponent exceeds three digits when the character string is normalized to the IEEE standard decimal format.	**strtod, fscanf, scanf, sscanf, atof**

Error No.	Message	Explanation	Functions to Set Error Numbers
1210	Overflow out of float	A float-type decimal value is out of range (overflow).	**strtod, fscanf, scanf, sscanf, atof**
1220	Underflow out of float	A float-type decimal value is out of range (underflow).	**strtod, fscanf, scanf, sscanf, atof**
1250	Overflow out of double	A double-type decimal value is out of range (overflow).	**strtod, fscanf, scanf, sscanf, atof**
1260	Underflow out of double	A double-type decimal value is out of range (underflow).	**strtod, fscanf, scanf, sscanf, atof**
1270	Overflow out of long double	A long double-type decimal value is out of range (overflow).	**fscanf, scanf**
1280	Underflow out of long double	A long double-type decimal value is out of range (underflow).	**fscanf, scanf**
1300	File not open	The file is not open.	**fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen**
1302	Bad file number	An **output** function was issued for an input file or **output** function is issued for input file.	**fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite**
1304	Error in format	An erroneous format was specified for an in input/output function using format.	**fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, perror**

APPENDIX

Appendix A Language and Standard Library Function Specifications of the C Compiler

This section shows the implementation dependent specifications of the C compiler that are not included in the C language specifications (in ANSI standard for the C programming language).

A.1 Language Specifications of the C Compiler

A.1.1 Compilation Specifications

Table A-1 Compilation Specifications

Item	C Compiler Specification
Error information when an error is detected	Refer to part IV, Error Messages

A.1.2 Environmental Specifications

Table A-2 Environmental Specifications

Item	C Compiler Specification
Actual argument for the main function	Not specified
Interactive I/O device configuration	Not specified

A.1.3 Identifiers

Table A-3 Identifier Specifications

Item	C Compiler Specification
Number of valid characters of internal identifiers not used for external linkage	The first 31 characters are valid
Number of valid characters of external identifiers used for external linkage	The first 31 characters are valid
Lowercase and uppercase character distinction in external identifiers used for external linkage	Lowercase characters are distinguished from uppercase characters.

Note: Two different identifiers with the same first 31 characters are considered to be identical.

Example:

(a) longnameabcdefghijklmnopqrstuvwx;
(b) longnameabcdefghijklmnopqrstuvwy;

Identifiers (a) and (b) are indistinguishable because the first 31 characters are the same.

A.1.4 Characters

Table A-4 Character Specifications

Item	C Compiler Specification
Elements of character set and codes used during program execution	ASCII character set Kanji used in host environment can be used for source program comment.
Shift state used for encoding multiple-byte characters	Shift state is not supported
The number of bits used to indicate a character sets during program execution	Eight bits are used for each character.
Correspondence between the program compilation character set and the execution	ASCII is used for both.
Extended representation that appears either in a character constant or a string literal and that is not defined in the language specifications	Characters and extended representation other than that specified by the language are not supported.
Character constant or wide character constant of two or more characters	The upper four characters of the character constant is valid, and the upper two characters of the wide character is valid. If a wide character of more than one character is specified, a warning error message is output.
locale specifications used to converting multiple-byte character to wide character	**locale** is not supported
Simple **char** having normal the value range same as **signed char** or **unsigned char**.	The same range as the **signed char**

A.1.5 Integer

Table A-5 Integer Specifications

Item	C Compiler Specification
Integer-type data representation and value	Table A-6 shows data representation and value.
Effect when an integer is too large to be converted into a **signed** integer-type value or **signed char**	The lower one or two bytes of the integer is used as the conversion result.
The result of bitwise operations on **signed** integers	**signed** value
Sign of the remainder for integer division	Same as the sign of the dividend
Effect of a right shift operation on the sign bit of signed integer-type data	The sign bit is unchanged by the shift operation.

Table A-6 Integer Types and Their Corresponding Data Range

Type	Range of Values	Data Size
char	–128 to 127	1 byte
signed char	–128 to 127	1 byte
unsigned char	0 to 255	1 byte
short	–32768 to 32767	2 bytes
unsigned short	0 to 65535	2 bytes
int	–2147483648 to 2147483647	4 bytes
unsigned int	0 to 4294967295	4 bytes
long	–2147483648 to 2147483647	4 bytes
unsigned long	0 to 4294967295	4 bytes

A.1.6 Floating-Point Numbers

Table A-7 Floating-Point Number Specifications

Item	C Compiler Specification
Data that can be represented as floating-point type and value	The **float**, **double**, and **long double** are provided as floating-point types. See section A.3, Floating-Point Number Specifications, for details on floating-point numbers (internal representation, conversion specifications, and operation specifications). Table A-8 shows the limits on representing floating-point numbers.
Data converted from **double** or **long double** to **float**	
Internal representation of floating-point data	

Table A-8 Limits on Floating-Point Numbers

	Limit	
Item	**Decimal** *1	**Internal Representation**
Maximum **float**	3.4028235677973364e+38f (3.4028234663852886e+38f)	7f7fffff
Positive minimum **float**	7.0064923216240862e-46f (1.4012984643248171e-45f)	00000001
Maximum **double** or **long double**	1.7976931348623158e+308 (1.7976931348623157e+308)	7fefffffffffffff
Positive minimum **double** or **long double**	4.9406564584124655e-324 (4.9406564584124654e-324)	0000000000000001

Note: *1. Limits on decimal is non-zero minimum value or maximum value not infinitive value. Values within ( ) indicate theoritical values.

A.1.7 Arrays and Pointers

Table A-9 Array and Pointer Specifications

Item	C Compiler Specification
Integer type required for array's maximum size (size_t)	**unsigned long**
Conversion from pointer-type data to integer-type data (Pointer-type data size ≥ Integer-type data size)	The lower byte of pointer-type data is used.
Conversion from pointer-type data to integer-type data (Pointer-type data size < Integer-type data size)	Extended with signs
Conversion from integer-type data to pointer-type data (Integer-type data size ≥ Pointer-type data size)	The lower byte of integer-type data is used.
Conversion from integer-type data to pointer-type data (Integer-type data size < Pointer-type data size)	Extended with signs
Integer type required for holding pointer difference between members in the same array **(ptrdiff_t)**	**long**

A.1.8 Register

Table A-10 Register Specifications

Item	C Compiler Specification
The maximum number of register variables that can be allocated to registers	7
Type of register variables that can be allocated to registers	**char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float,** and pointers

A.1.9 Structure, Union, Enumeration, and Bit Field Types

Table A-11 Specifications for Structure, Union, Enumeration, and Bit Field Types

Item	C Compiler Specification
Effect of setting a union member and referencing a union member using another member whose data type is different	Reference is possible but the referred value is not guaranteed.
Structure member alignment	Structures consisting of **char** members are aligned in 1-byte units, while structures consisting of **short** members are aligned in 2-byte units. Structures consisting of any other members are aligned in 4-byte units.*1
Sign of an **int** bit field	Assumed to be **signed int**
Allocation order of bit fields in **int** area	Beginning from the high order bit to low order bit.*2
Result when a bit field has been allocated in an **int** area and the next bit field to be allocated is larger than the remaining **int**	The next bit field is allocated to the next **int** area.*2
Type specifier allowed for bit field	**char, unsigned char, short, unsigned short, int, unsigned int, long,** and **unsigned long**
Integer describing enumeration	**int**

Notes: *1. See section 2.2 (2), Aggregate Data, in part II for details on structure member allocation.

*2. See section 2.2 (3), Bit Fields, in part II for details on bit field allocation.

A.1.10 Modifier

Table A-12 Modifier Specifications

Item	C Compiler Specification
volatile data access type	Not specified

A.1.11 Declarations

Table A-13 Declaration Specifications

Item	C Compiler Specification
Number of types that can qualify the basic types (pointer, array, and function)	Up to 16 types can be specified.

(a) Example of counting the number of types that qualify the basic types

Examples:

(i) int a;

a is **int** (basic type) and the number of types that qualify the basic type is zero.

(ii) char *f( );

f is a function type that returns pointer to **char** (basic type). The number of types that qualify the basic type is two.

A.1.12 Statement

Table A-14 Statement Specifications

item	C Compiler Specification
The number of case label specified by a switch statement	Up to 511 labels can be specified.

A.1.13 Preprocessor

Table A-15 Preprocessor Specifications

Item	C Compiler Specification
Correspondence between single character constant and execution environment characters in the conditional compilation	Character strings in the preprocessor statement match the execution environment characters
Reading an include file	The file within < > is read from a path specified by the **include** option. (Defalut: The path specified by environment variable SHC_LIB)
Supporting an include file whose name is enclosed in a pair of double quotation marks	The C compiler supports include files whose names are delimited by double quotation marks. The C compiler reads these include files from the current directory. If the include files are not in the current directory, the C compiler reads them from the directory specified in advance.
Source file character string correspondence (blank character in a character string after macro expansion)	Strings of blanks are expanded as one blank character.
#pragma directive operation	**#pragma interrupt** is supported.*1
Value of __DATE__, __TIME__	Data depending on the host machine timer when the compilation starts.

Note: *1. See section 3.1, Interrupt Functions, in part II for details on #pragma interrupt specifications.

A.2 C Library Function Specifications

This section explains the specifications for C library functions declared in standard include files. Refer to the include file for the actual macro names defined in a standard include file.

A.2.1 stddef.h

Table A-16 stddef.h Specifications

Item	C Compiler Specification
Value of macro **NULL**	The value 0 of pointer to **void**
Contents of macro **ptrdiff_t**	**long**

A.2.2 assert.h

Table A-17 assert.h Specifications

Item	C Compiler Specification
Information output and terminal operation of **assert.h**	See (a) for the format of output information. The program outputs information and then calls the **abort** function to stop the operation.

(a) The following message is output when the expression is 0 for assert (expression):
`Assertion Failed:` <expression> `File` <file-name>, `Line` <line-number>

A.2.3 ctype.h

Table A-18 ctype.h Specifications

Item	C Compiler Specification
The character set for which the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions	Table A-19 shows the character set that results in a true return value.

Table A-19 Set of Characters that Returns True

Function Name	Characters That Become True
isalnum	'0' to '9', 'A' to 'Z', 'a' to 'z'
isalpha	'A' to 'Z', 'a' to 'z'
iscntrl	'\0' to '\037', '\177'
islower	'a' to 'z'
isprint	'\40' to '\176'
isupper	'A' to 'Z'

A.2.4 math.h

Table A-20 math.h Specifications

Item	C Compiler Specification
Value returned by a mathematical function if an input parameter is out of the range	Returns a nonnumeric value
Is errno set to the value of macro **ERANGE** if an underflow error occurs in a mathematical function?	Yes, it is set.
Does a range error occur if the 2nd parameter in the fmod function is 0	A range error occurs

Note: **math.h** defines macro names **EDOM** and **ERANGE** that indicates a standard library error number.

A.2.5 stdio.h

Table A-21 stdio.h Specifications

Item	C Compiler Specification
Is a return character indicating input text end required?	Not specified. Depends on the low-level interface routine specifications.
Is a blank character immediately before the carriage return read?	
Number of **NULL** characters added to data written to binary file	
Initial value of file position specifier in addition mode	
Is a file data lost following text file output?	
File bufferring specifications	
Is a file with file length 0 exists?	
File name configuration rule	
Can the same files be opened simultaneously?	
Output data representation of the %p format conversion in the **fprintf** function	Hexadecimal representation
Input data representation of the %p format conversion in the fscan function, the meaning of – in the fscanf function	Hexadecimal representation If – does not follow ^, indicates the range between the previous and following characters.
Value of **errno** specified by **fgetpos** and **ftell** functions	The **fgetpos** function is not supported. The **ftell** function does not specify the errno value. The errno value is determined depending on the low-level interface routine.
Output format of messages generated by the **perror** function	See (a) below for the output message format.
calloc, **malloc**, or **realloc** function operation when the size is 0	0 byte area is allocated.

(a) Messages generated by a **perror** function follow this format:
<string-literal> : <error-message corresponding to the error number indicated by errno>

(b) Table A-22 shows the format used to indicate infinity and not a number for **floating-point** numbers when using the **printf** or **fprintf** function.

Table A-22 Infinity and Not a Number

Value	Format
Positive infinity	++++++
Negative infinity	------
Not a number	******

A.2.6 string.h

Table A-23 string.h Specifications

Item	C Compiler Specification
Error message returned by the **strerror** function	See part IV, section 2, Standard Library Error Messages.

A.2.7 Not Supported Library

Table A-24 lists libraries in the C language specifications not supported by the C compiler

Table A-24 Libraries Not Supported by the C Compiler

Header File	Library Name
signal.h	signal, raise
stdio.h	remove, rename, tmpfile, tmpnam
stdlib.h	getenv, system
time.h	clock, difftime, time, asctime, ctime, gmtime, localtime

A.3 Floating-Point Number Specifications

A.3.1 Internal Representation of Floating-Point Numbers

The internal representation of floating-point numbers follows the standard IEEE format. This section explains this standard.

Internal Representation Format: **float** is represented in IEEE single precision (32 bits), **double** and **long double** are represented in IEEE double precision (64 bits).

Internal Representation Structure: Figure A-1 shows the structure of **float**, **double**, and **long double** in internal representation.

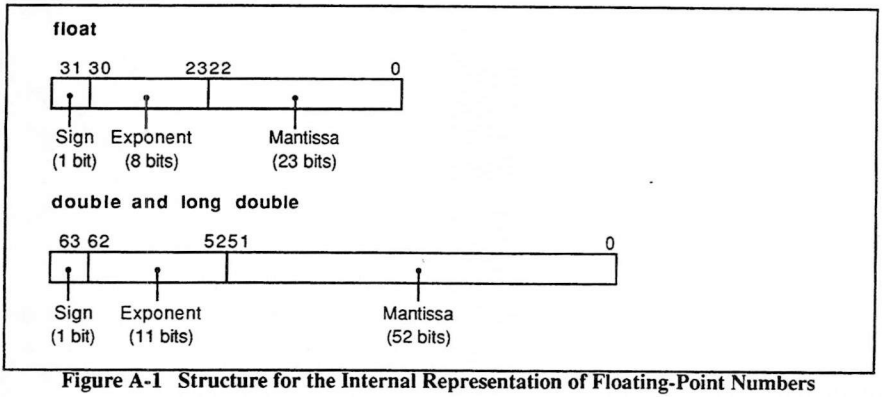


Figure A-1 Structure for the Internal Representation of Floating-Point Numbers

The elements of the structure have the following meanings.

(i) Sign
This indicates the sign of a floating-point number. Positive and negative are represented by 0 and 1, respectively.

(ii) Exponent
This indicates the exponent of a floating-point number as a power of two.

(iii) Mantissa
This determines the significant digits of a floating-point number.

Types of Values: Floating-point numbers can represent infinity in addition to general real numbers. The rest of this section explains the types of values that can be represented by floating-point numbers.

(i) Normalized Number
The exponent is not 0 or the maximum. A normalized number represents a general real number.

(ii) Denormalized Number
The exponent is 0 and the mantissa is not 0. A denormalized number is a real number whose absolute value is very small.

(iii) Zero
The exponent and mantissa are both 0. Zero represents the value 0.0.

(iv) Infinity
The exponent is the maximum and mantissa is 0.

(v) Not a Number
The exponent is the maximum and the mantissa is not 0. This is used to represent an operation result that is undefined (such as 0.0/0.0, ∞/∞, $\infty - \infty$).

Table A-25 shows the conditions used to determine values represented by floating-poi.nt numbers.

Note: A denormalized number represents a floating-point number whose absolute value is so small that it cannot be represented as a normalized number. Denormalized numbers have less significant digits than normalized numbers. The significant digits of a result are not guaranteed if either the operation result or an intermediate result is a denormalized number.

Table A-25 Types of Values Represented by Floating-Point Numbers

	Exponent		
Mantissa	**0**	**Other than 0 or Maximum**	**Maximum**
0	0	Normalized number	Infinity
Other than 0	Denormalized number		Not a number

A.3.2 float

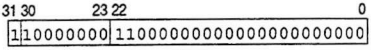
float is internally represented as 1 sign bit, 8 exponent bits, and 23 mantissa bits.

Normalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is a number from 1 to 254 ($2^8 - 2$). From this value 127 is subtracted and the result is used as the actual exponent. The range of actual exponents is –126 to 127. The mantissa is a value from 0 to $2^{23} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{23}) is 1 and a decimal point follows it.

Value represented by a normalized number:

$$(-1)^{<sign>} \times 2^{<exponent> - 127} \times (1 + <mantissa> \times 2^{-23})$$

Example:



Sign: –

Exponent: $10000000_{(2)} - 127 = 1$ ((2) indicates decimal data throughout this manual.)

Mantissa: $1.11_{(2)} = 1.75$

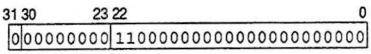
Value: $-1.75 \times 2^1 = -3.5$

Denormalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is 0 which makes the actual exponent equal to –126. The mantissa is a value from 1 to $2^{23} - 1$. For an actual mantissa, it is assumed that a highest order bit (2^{23}) is 0 and a decimal point follows it.

Value represented by a denormalized number:

$$(-1)^{<sign>} \times 2^{-126} \times (<mantissa> \times 2^{-23})$$

Example:



Sign: +

Exponent: –126

Mantissa: $0.11_{(2)} = 0.75$

Value: 0.75×2^{-126}

Zero: The sign bit is either 0 (positive) or 1 (negative), (i.e., there are two distinct zero values, +0.0 and –0.0). The exponent and mantissa are 0. Both +0.0 and –0.0 represent 0.0. See appendix A.3.4, Floating-Point Operation Specifications, for differences in each operation depending on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) (i.e., $+\infty$ and $-\infty$ can be represented). The exponent is 255 ($2^8 - 1$). The mantissa is 0.

Not a Number: The exponent is 255 ($2^8 - 1$) and the mantissa is not equal to 0.

Note: The sign of a not a number is arbitrary and the value of the mantissa is not limited (except that it may not be equal to 0).

A.3.3 **double** and **long double**

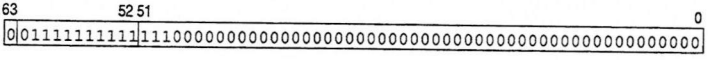
A **double** or **long double** is represented as 1 sign bit, 11 exponent bits, and 52 mantissa bits.

Normalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is a number from 1 to 2046 ($2^{11} - 2$). From this value 1023 is subtracted and the result is used as the actual exponent. The range of actual exponents is –1022 to 1023. The mantissa is a value from 0 to $2^{52} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{52}) is 1 and a decimal point follows it.

Value represented by a normalized number:

$$(-1)^{<sign>} \times 2^{<exponent> - 1023} \times (1 + <mantissa> \times 2^{-52})$$

Example:



Sign: +

Exponent: $1111111111_{(2)} - 1023 = 0$

Mantissa: $1.111_{(2)} = 1.875$

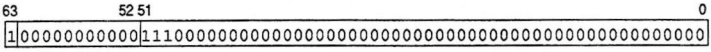
Value: $1.875 \times 2^0 = 1.875$

Denormalized Number: The sign bit is either 0 (positive) or 1 (negative). The exponent is 0 which makes the actual exponent equal to –1022. The mantissa value is from 1 to $2^{52} - 1$. For an actual mantissa, it is assumed that the highest order bit (2^{52}) is 0 and a decimal point follows it.

Value represented by a denormalized number:

$(-1)^{<sign>} \times 2^{-1022} \times (<mantissa> \times 2^{-52})$

Example:



Sign: –

Exponent: –1022

Mantissa: $0.111_{(2)} = 0.875$

Value: 0.875×2^{-1022}

Zero: The sign bit is either 0 (positive) or 1 (negative) (i.e., there are two distinct zero values +0.0 and –0.0). The exponent and mantissa are 0. Both +0.0 and –0.0 represent 0.0. See appendix A.3.4, Floating-Point Operation Specifications, for differences in each operation depending on the sign.

Infinity: The sign bit is either 0 (positive) or 1 (negative) (i.e., $+\infty$ and $-\infty$ can be represented). The exponent is 2047 ($2^{11} - 1$). The mantissa is 0.

Not a Number: The exponent is 2047 ($2^{11} - 1$) and the mantissa is not equal to 0.

Note: The sign of a not a number is arbitrary and the value of the mantissa is not limited (except that it may not be equal to 0).

A.3.4 Floating-point Operation Specifications

This section explains the floating-point arithmetic used in C language functions. It also gives the specifications for converting between the decimal representation and the internal representation of floating-point numbers generated during C compiler or standard library function processing.

Arithmetic Operation Specifications:

(i) Result Rounding

If the precise result of a floating-point operation exceeds the significant digits of the internally represented mantissa, the result is rounded as follows:

① The result is rounded to the nearest internally representable floating-point number.

② If the result is directly between the two nearest internally representable floating-point numbers, the result is rounded so that the lowest bit of the mantissa becomes 0.

(ii) Overflow and Underflow Handling

Invalid operations, overflows and underflows resulting from numeric operations are handled as follows:

① For an overflow, positive or negative infinity is used depending on the sign of the result.

② For an underflow, positive or negative zero is used depending on the sign of the result.

③ An invalid operation is assumed when: (i) infinity is added to infinity and each infinity has a different sign, (ii) infinity is subtracted from infinity and each infinity has the same sign, (iii) zero is multiplied by infinity, (iv) zero is divided by zero, or (v) infinity is divided by infinity. In each case, the result is not a number.

④ In any case, the variable errno is set to the error number corresponding to the error. See part IV, Error Messages, section 2, C Library Error Messages, for the error number.

Note: Operations are performed with constant expressions at compile time. If an overflow, underflow, or invalid operation is detected during these operations, a warning-level error occurs.

(iii) Special Value Operations

More about special value (zero, infinity, and not a number) operations:

① The result is positive zero if positive zero and negative zero are added.

② If zero is subtracted from zero and both zeros have the same sign, the result is positive zero.

③ The operation result is always a not a number if one or both operands are not a numbers.

④ Positive zero is equal to a negative zero for relational operations.

⑤ If one or both operands are not a numbers in a relational or equivalence operation, the result of != is always true and all other results are false.

Conversion between Decimal Representation and Internal Representation: This section explains the conversion between floating-point constants in a source program and floating-point constants in internal representation. The conversion between decimal representation and internal representation of ASCII string literal floating-point numbers by library functions is also explained.

(i) To convert a floating-point number from decimal representation to internal representation, the floating-point number in decimal representation is first converted to a floating-point number in normalized decimal representation. A floating-point number in normalized decimal representation is in the format $\pm M \times 10^{\pm N}$. The following ranges of M and N are used:

① For normalized **float**

$0 \leq M \leq 10^9 - 1$

$0 \leq N \leq 99$

② For normalized **double** and **long double**

$0 \leq M \leq 10^{17} - 1$

$0 \leq N \leq 999$

An overflow or underflow occurs if a floating-point number in decimal representation cannot be normalized. If a floating-point number in normalized decimal representation contains too many significant digits, as a result of the conversion, the lower digits are discarded. In the above cases, a warning-level error occurs at compile time and the variable errno is set equal to the corresponding error number at run time.

To convert a floating-point number from decimal representation to normalized decimal representation, the length of the original ASCII string literal must be less than or equal to 511 characters. Otherwise, an error occurs at compile time and the variable **errno** is set equal to the corresponding error number at run time.

To convert a floating-point number from internal representation to decimal representation, the floating-point number is first converted from internal representation to normalized decimal representation. According to a specified format, the result is then converted to an ASCII string literal.

(ii) Conversion between Normalized Decimal Representation and Internal Representation

If the exponent of a floating-point number to be converted between decimal representation and internal representation is too large or too small, a precise result cannot be obtained. This section explains the range of exponents for precise conversion and the error that results from exceeding the range.

a) Range of Exponents for Precise Conversion

Rounding as explained in the description, Result Rounding, in appendix A.3 4, Floating-point Operation Specifications, is performed precisely for floating-point numbers whose exponents are in the following ranges:

① For **float** : $0 \leq M \leq 10^9 - 1,\ 0 \leq N \leq 13$

② For **double** and **long double**: $0 \leq M \leq 10^{17} - 1,\ 0 \leq N \leq 27$

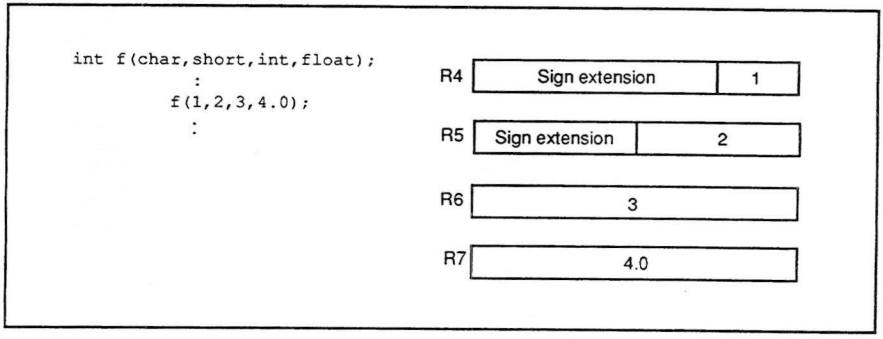
An overflow or underflow will not occur if the exponent is within the proper ranges.

b) Conversion and Rounding Error

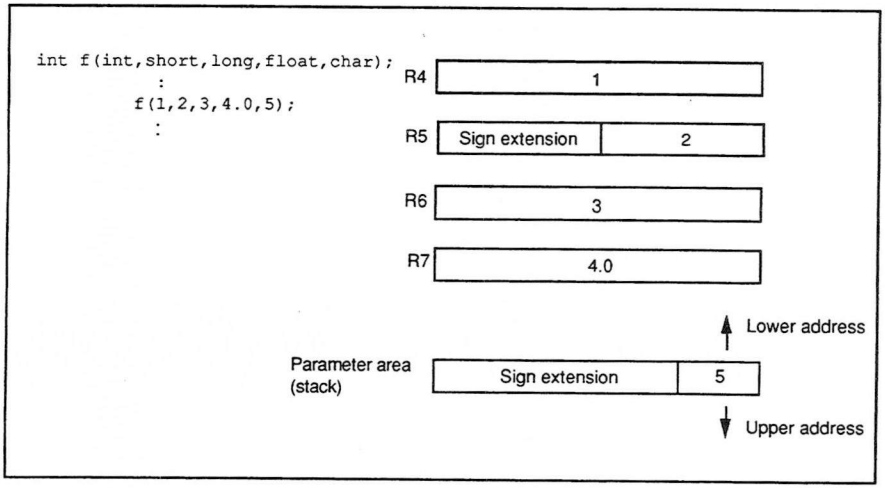
The difference between, (i) the error occurring when the exponent outside the proper range is converted, and (ii) the error occurring when the value is precisely rounded, does not exceed the result of multiplying the lowest significant digit by 0.47. If an exponent outside the proper range is converted, an overflow or underflow may occur. In such a case, a warning-level error occurs at compile time and the variable errno is set equal to the corresponding error number at run time.

Appendix B Parameter Allocation Example

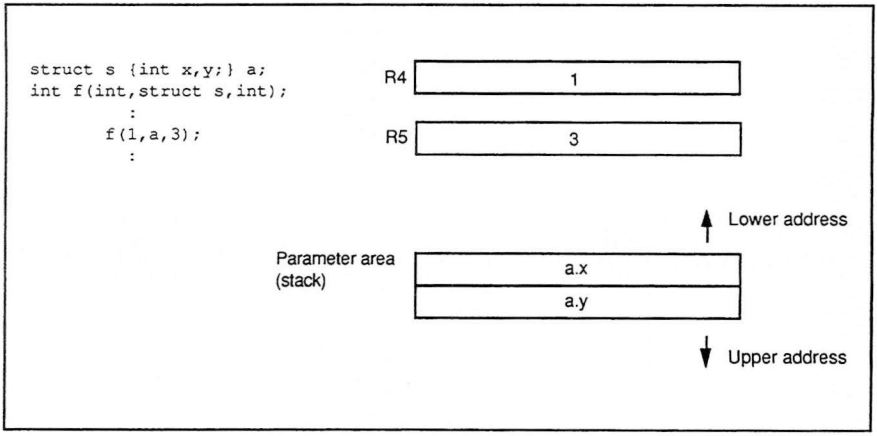
Example 1: Register parameters are allocated to registers from R4 to R7 depending on the order of declaration.



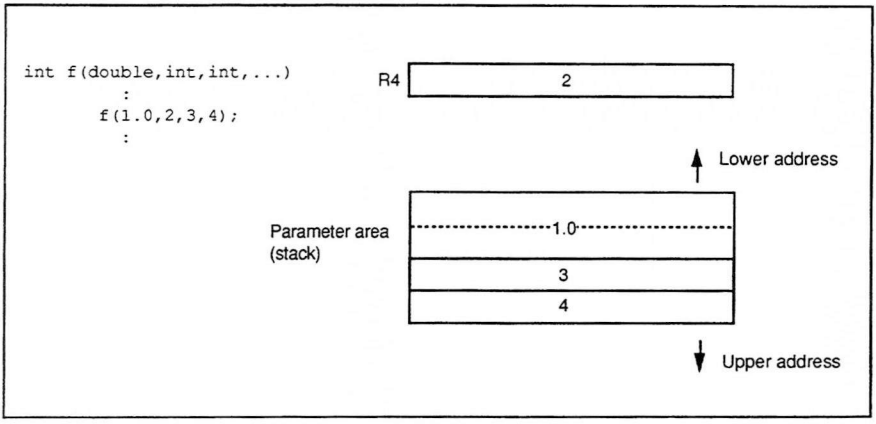
Example 2: Parameters which could not be allocated to registers from R4 to R7 are allocated to the stack area as shown below. If a **char (unsigned)** or **short (unsigned)** type parameter is allocated to a parameter area on a stack, it is extended to a 4-byte area.



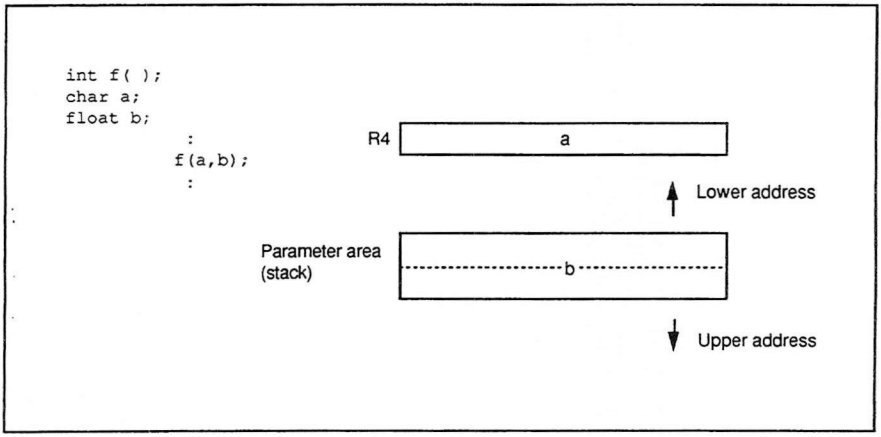
Example 3: Parameters having a type that cannot be allocated to registers from R4 to R7 are allocated to the stack area.



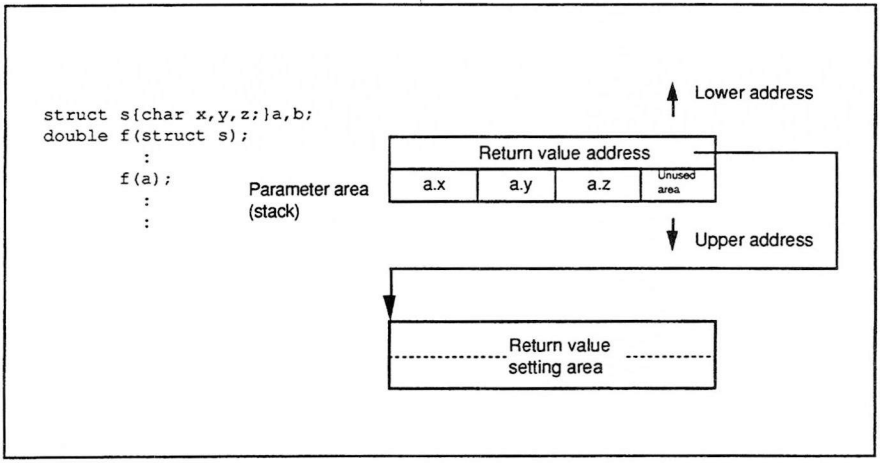
Example 4: If a function whose number of parameters changes is specified by prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding paramters are allocated to a stack.



Example 5: If no prototype is declared, **char** and **float** types are extended to **int** and **double** types, respectively.



Example 6: If a value returned by a function exceeds 4 bytes, or is a structure type, a return value is specified just before parameter area. If structure size is not a multiple of four, an unused area is generated.



Appendix C Usage of Registers and Stack Area

This section describes how to use registers and stack area by the C compiler. The user does not need to note how to use this area, because registers and stack area used by a function are operated by the C compiler. Figure C-1 shows the usage of registers and stack area.

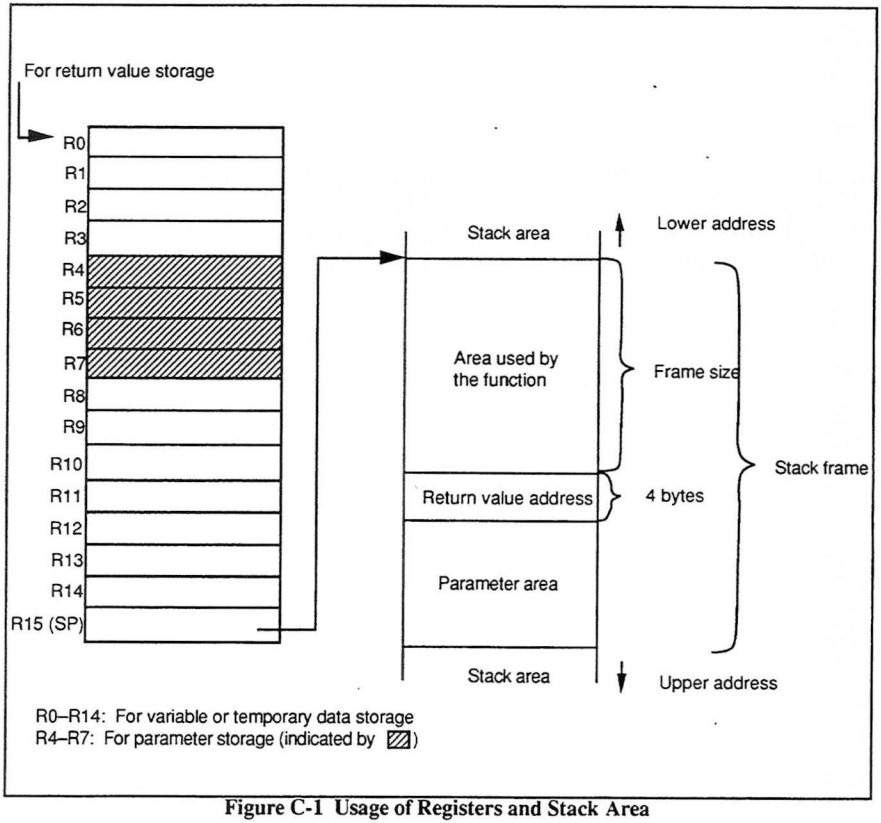


Figure C-1 Usage of Registers and Stack Area

Appendix D Creating Termination Functions

D.1 Creating Library onexit Function

This section describes how to create library onexit function that defines termination routines. The onexit function defines a function address, which is passed as a parameter, in the termination routine table. If the number of defined functions exceeds the limit value (assumed to be 32 in the following example), or if the same function is defined twice or more, NULL is returned. Otherwise, value other than NULL is returned. In the following example, an address in which a function is defined is returned. An example of onexit routine is shown below.

Example:

```
#include <stdlib.h>
typedef void *onexit_t ;

int _onexit_count=0 ;
onexit_t (*_onexit_buf[32])(void) ;

extern  onexit_t onexit(onexit_t (*)(void)) ;

onexit_t onexit(f)
onexit_t (*f)(void) ;
{
      int i;

      for(i=0; i<_onexit_count ; i++)      /*Checks if the same function has been defined*/
            if(_onexit_buf[i]==f)
                  return NULL ;
     if (_onexit_count==32)                /*Checks if the No. of defined functions exceed limit*/
           return NULL ;
     else{
          _onexit_count++ ;
          _onexit_buf[_onexit_count]=f ;  /*Defines the function address*/
          return &_onexit_buf[_onexit_count];
     }
}
```

D.2 Creating exit Function

This section describes how to create exit function that terminates program execution. Note that the exit function must be created according to the user system specifications refereing to the following example, because how to terminate a program differs depending on the user system.

The exit function terminates C program execution based on the termination code returned as a paramter and then returns to the environment at program initiation. Returning to the environment at program initiation is achieved by the following two steps:

(1) Sets a termination code in an external variable

(2) Returns to the environment that is saved by the setjmp function immediately before calling the main function

An example of the exit function is shown below.

```
#include <setjmp.h>
#include <stddef.h>

typedef void * onexit_t ;
extern int _onexit_count ;
extern onexit_t (*_onexit_buf[32])(void) ;

extern jmp_buf _init_env ;
extern int _exit_code ;

extern void _CLOSEALL();
extern void exit(int);

void exit(code)
int code ;
{
      int i;

      _exit_code=code ;                 /*Sets return code to _exit_code */

      for(i=_onexit_count-1; i>0; i--) /*Sequencially executes functions defined by onexit*/
            (*_onexit_buf[i])();

      _CLOSEALL();                      /*Closes all files opened*/

      longjmp(_init_env, 1) ;           /*Returns to the environment saved by the setjmp*/
}
```

Note: To return to the environment before program execution, create the **callmain** function and call the callmain function instead of calling the main function from the init routine as shown below.

```
#include <setjmp.h>

jmp_buf _init_env;
int     _exit_code;

void callmain()
{

   /*Saves current environment by setjmp function and calls the main function */

   /*Terminates C program if a termination code is returned from the exit function*/

   if(!setjmp(_init_env))
         _exit_code=main();
}
```

D.3 Creating abort Routine

To terminate the routine abnormally, the program must be terminated by a abort routine prepared according to the user system specifications. The following shows an example of abort routine in which an error message is output to the standard output device, closes all files, enters endless loop, and wait for reset.

Example:

```
#include <stdio.h>

extern void abort(void);

extern void _CLOSEALL();

void abort()
{
      printf("program is abort !!\n"); /*Outputs message     */

      _CLOSEALL();                     /*Closes all files    */

      while(1);                        /*Enters endless loop */

}
```

Appendix E Examples of Low-Level Interface Routine

```
/*****************************************************************************/
/*                        lowsrc.c:                                          */
/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*         SH-series simulator debugger interface routine                    */
/*   - Only standard I/O files (stdin, stdout, stderr) are supported         */
/*****************************************************************************/
#include <string.h>

/* file number */

#define STDIN  0                        /* Standard input (console)          */
#define STDOUT 1                        /* Standard output (console)         */
#define STDERR 2                        /* Standard error output (console)   */

#define FLMIN  0                        /* Minimum file number               */
#define FLMAX  3                        /* Maximum number of files           */

/* file flag */

#define O_RDONLY 0x0001                 /* Read only                         */
#define O_WRONLY 0x0002                 /* Write only                        */
#define O_RDWR   0x0004                 /* Both read and write               */

/* special character code */

#define CR 0x0d                         /* Carriage return                   */
#define LF 0x0a                         /* Line feed                         */

/* size of area managed by sbrk */

#define HEAPSIZE 1024

/*****************************************************************************/
/* Declaration of reference function                                         */
/* Reference of assembly program in which the simulator debugger input or    */
/* output characters to the console                                          */
/*****************************************************************************/
extern void charput(char);              /* One character input               */
extern char charget(void);              /* One character output              */

/*****************************************************************************/
/* Definition of static variable:                                            */
/* Definition of static variables used in low-level interface routines       */
/*****************************************************************************/

char flmod[FLMAX];                      /* Open file mode specification area */

static  union  {
                long  dummy ;           /* Dummy for 4-byte boundary         */
                char heap[HEAPSIZE]; /* Declaration of the area managed by sbrk */
 }heap_area ;

static  char  *brk=(char *)&heap_area;/* End address of area assigned by sbrk */
```

```
/*****************************************************************************/
/*       open:file open                                                      */
/*          Return value:File number (Pass)                                  */
/*                       -1          (Failure)                               */
/*****************************************************************************/
int open(char *name,                        /* File name                     */
     int mode)                              /* File mode                     */
{
    /* Check mode depending on file name and return file numbers             */

     if(strcmp(name,"stdin")==0){           /* Standard input file           */
           if((mode&O_RDONLY)==0)
                    return -1;
           flmod[STDIN]=mode;
           return STDIN;
     }

     else if(strcmp(name,"stdout")==0){     /* Standard output file          */
           if((mode&O_WRONLY)==0)
                    return -1;
           flmod[STDOUT]=mode;
           return STDOUT;
     }

     else if(strcmp(name,"stderr")==0){     /* Standard error file           */
           if((mode&O_WRONLY)==0)
                    return -1;
           flmod[STDERR]=mode;
           return STDERR;
     }

     else
           return -1;                       /* Error                         */
}


/*****************************************************************************/
/*   close:File close                                                        */
/*         Return value:0 (Pass)                                             */
/*                      -1 (Filure)                                          */
/*****************************************************************************/
int close(int fileno)                       /* File number                   */
{
     if(fileno<FLMIN || FLMAX<fileno)       /* File number range check       */
           return -1;

     flmod[fileno]=0;                       /* File mode reset               */
     return 0;
}
```

```
/*****************************************************************************/
/* read:Data read                                                            */
/*      Return value:Number of read characters (Pass)                        */
/*                   -1                        (Failure)                     */
/*****************************************************************************/
int read(int  fileno,                       /* File number                   */
        char *buf,                          /* Destination buffer address    */
        unsigned int  count)                /* Number of read characters     */
{
      unsigned int i;

   /*Check mode according to file name and stores each character in buffer*/

      if(flmod[fileno]&O_RDONLY||flmod[fileno]&O_RDWR){
            for(i=count; i>0; i--){
                  *buf=charget();
                  if(*buf==CR)          /*Line feed character replacement*/
                        *buf=LF;
                  buf++;
            }
            return count;
       }
       else
            return -1;
}

/*****************************************************************************/
/* write:Data write                                                          */
/*       Return value:Number of write characters (Pass)                      */
/*                    -1                         (Failure)                   */
/*****************************************************************************/
int write(int  fileno,                      /* File number                   */
         char *buf,                         /* Destination buffer address    */
         unsigned int  count)               /* Number of write characters    */
{
       unsigned int  i;
       char c;

       /* Check mode according to file name and output each character     */

       if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
             for(i=count; i>0; i--){
                   c=*buf++;
                   charput(c);
             }
             return count;
       }
       else
             return -1;
}
```

```
/*****************************************************************************/
/* lseek:Definition of file read/write position                               */
/*      Return value:Offset from the top of file read/write position(Pass)*/
/*                   -1              (Failure)                               */
/*      (lseek is not supported in the console input/output)                 */
/*****************************************************************************/
long lseek(int  fileno,                    /* File number                    */
           long offset,                    /* Read write position            */
           int  base)                      /* Origin of offset               */
{
     return -1;
}

/*****************************************************************************/
/*     sbrk:Data write                                                       */
/*          Return value:Start address of the assigned area (Pass)           */
/*                    -1                                    (Failure)      */
/*****************************************************************************/
char  *sbrk(unsigned long size)            /* Assigned area size             */

      char  *p ;

      if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size                */
           return (char *)-1 ;

      p=brk ;                              /* Area assignment                */
      brk += size ;                        /* End address update             */
      return p ;
}
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                          lowlvl.src                                    |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;            SH-series simulator debugger interface routine              |
;                      -Input/output one character-                      |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          .EXPORT     _charput
          .EXPORT     _charget
SIM_IO:   .EQU            H'0080        ;Specifies TRAP_ADDRESS

          .SECTION    P, CODE, ALIGN=4

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: One character output                                         |
;          C program interface: charput(char)                             |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
          MOV.L       A_PARM, R1
          MOV         R4, R0            ;Specifies data
          MOV.B       R0, @(3, R1)
          MOV         #H'21, R0         ;Specifies function code
          MOV.B       R0, @R1
          MOV.L       A_FILENO, R0      ;Specifies file number
          MOV.B       @R0, R0
          MOV.B       R0, @(2, R1)
          MOV         R1, R0            ;Specifies parameter block address
          TRAPA       #SIM_IO
          NOP
          RTS
          NOP
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; _charget: One character input                                           |
;       C program interface: char charget(void)                          |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charget:
          MOV.L        A_PARM, R1
          MOV          #H'20, R0        ;Specifies function code
          MOV.B        R0, @R1
          MOV.L        A_FILENO, R0     ;Specifies file number
          MOV.B        @R0, R0
          MOV.B        R0, @(2, R1)
          MOV          R1, R0           ;Specifies parameter block address
          TRAPA        #SIM_IO
          NOP
          MOV.L        A_PARM, R1
          MOV.B        @(3, R1), R0     ;References data
          RTS
          NOP

          .ALIGN       4
A_PARM:   .DATA.L      PARM             ;Parameter block address
A_FILENO: .DATA.L      FILENO           ;File number area address

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                        I/O buffer definition                           |
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .SECTION    B,DATA,ALIGN=4

PARM:     .RES.L      1              ; Parameter block area
FILENO:   .RES.B      1              ; File number area

          .END
```

Appendix F ASCII Codes

UPPER 4 BITS / LOWER 4 BITS	0	1	2	3	4	5	6	7
0	NUL	LE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Index

Numbers
Decimal and internal representation 163

A
abort routine (termination routine) 171
Aggregate-type data 27
Alignment 24, 26
Area size calculation 58, 63
  Heap area 66
  Stack area size calculation 58
  Stack area 63
Array type 27
ASCII codes 177
asmcode (suboption) 9

B
Bit field 28, 150
bss (suboption) 9

C
C compiler listings 12
char-type data 26, 146
C library function 60, 153
  Error messages output for the C library functions 140
C library function execution environment setting 73
  Closing files 79
  Initialization 75
  Initializing C library functions 76
  Initializing sections 76
  Low-level interface routines 80
  Program configuration 73
  Vector table setting 74
close routine (low-level interface routine) 84
code (option) 9
Coding notes 51
command line specification (C compiler listings) 17
Compiler option 9
  C compiler options 5, 6
  C compiler option listings 9
  Option combinations 11
const (suboption) 9
Constant area 24
cpu (option) 9
Creating low-level interface routines 80
Creating termination function 169
  Creating abort routine 171
  Creating exit function 170
  Creating library onexit function 169

D
data (suboption) 9
debug (option) 9
Debug information 9, 11, 54
define (option) 9
Denormalized number 158
double 26, 148, 160
Dynamic area 24, 63
  Dynamic area allocation 63
  Heap area 24, 66
  Stack area 24, 63
Dynamic area size calculation 63

E
errno 76
Error levels (error message levels) 92
  Error level 92
  Fatal level 92
  Internal level 92
  Warning level 92
Error messages 91
  C compiler action and programmer response for each error level 92
  Error levels 92
  Error message format 91
  Error message output for the C compiler library functions 140
Execution environment setting 67
  Initialization 69
  Program configuration 67
  Section initialization 70
  Vector table setting 68
exit function (termination routine) 170
expansion (suboption) 9
Exponent 157
Extended specifications 43
  Interrupt functions 43
  Intrinsic functions 47
External identifier 32

F
Fatal level (error message level) 92
File close 79
File extension 8
File I/O operation 57, 77, 80
close routine 84
lseek routine 87
open routine 82
read routine 85
sbrk routine 88
write routine 86
File naming 8
float 26, 148, 159
Floating-point number specifications 148, 157
Denormalized number 158
Exponent 157
Infinity 158
Limits on floating-point numbers 148
Internal representation format 157
Mantissa 157
Normalized number 158
Not a number 158
Sign 157
Types of values 158
Floating-point operation specifications 162
Conversion between decimal representation and internal representation 163
Invalid operation 162
Overflow 162
Result rounding 162
Special value operations 162
Underflow 162
Frame size 15, 64, 168
Function call interface 34

G
Global base register (GBR) 47, 48

H
Heap area 24, 66
help (suboption) 9
How to invoke the C compiler 5

I
include (option) 9
include (suboption) 9
Include file 8
Reading an include file 152
Standard include file 4
Infinity 158
int 26, 147
Initialization 67, 69, 74, 75
Initialized data area 24
Internal data representation 26
Internal errors 92
Internal representation 26, 157
Internal representation of scalar-type data 26
Interrupt functions 43
Stack switching specification 43, 44
Trap-instruction return specification 43, 44
Intrinsic functions 47
Invalid operation 162
I/O operation 57
Concept of I/O operations 80
Low-level interface routine 80
Routine for standard library function 77

J

K

L
Language specifications 145
Arrays and pointers 149
assert.h 153
Characters 146
C library function specifications 153
Compilation specifications 145
ctype.h 154
Declarations 151
Environmental specifications 145
Floating-point numbers 148
Floating-point number specifications 157
Identifiers 145
Integer 147
Integer types and their corresponding data range 147
Limits on floating-point numbers 148
math.h 154
Modifier 150
Not supported library 156
Preprocessor 152
Register 149
Statement 151
stddef.h 153
stdio.h 155
string.h 156
Structure, union, enumeration, and bit field types 150
length (suboption) 9

Library 4
C library function 4, 60
Error messages output for the C library functions 140
Initializing C library functions 76
Low-level interface routine 80
Not supported library 156
Run time routine 4, 60
Standard library file 4
Limitation 21, 22
Linkage with assembly programs 31
Allocating deallocating stack frames 34
External identifier reference 32
Function call interface 34
Registers 35
Setting and referencing parameters and return values 37
Stack pointer 34
listfile (option) 9
Listing 12
command line specification 17
Object information listing 15
Source listing 13
Statistics information 16
Structure of C compiler listings 12
long double 26, 148, 160
Low-level interface routines 80
close routine 84
Examples of low-level interface routine 172
lseek routine 87
open routine 82
read routine 85
sbrk routine 88
write routine 86
lseek routine (low-level interface routine) 87

M
machinecode (suboption) 9
Macro name definition 9
Mantissa 157
Memory allocation 57
sbrk routine 80, 88
Memory area allocation 58
Area size calculation 58, 63
Example: Memory area allocation and address specification at program linkage 61
Dynamic area allocation 63
Initialized data area allocation 61
ROM and RAM allocation 61
Static area allocation 58

N
Non-initialized data area 24
Normalized number 158
Not a number 158
Notes on programming 51
Coding notes 51
Notes on programming development 54

O
object (suboption) 9
objectfile (option) 9
Object information (C compiler listings) 12, 15
onexit function (termination routine) 169
open routine (low-level interface routine) 82
optimize (option) 9
Optimization level 9
Overflow 162

P
Parameter 37
Parameter allocation example 165
Parameter area allocation 39
Passing parameters 37
Rules on type conversion 37
Stack parameter area 39
Storage registers 41
#pragma interrupt 43
program (suboption) 9
Program area 24
Program configuration 67, 73
Program development notes 54
PR register 36

Q

R
read routine 85
Register 149
Parameter register 41, 168
Return value storage register 42
Rules on changes in registers after a function call 35
Usage of registers 168
Reserved words 80
Result rounding 162

Return value 37
General rules concerning return values 37
Return value address 42
Return value setting location 42
Return value storage register 42
ROM (linkage editor option) 62
ROM and RAM allocation 61
ROM option 62
Run time routines 4, 60

S
sbrk routine (low-level interface routine) 88
section (option) 9
Section 24
Constant area 22
Initialized data area 22
Non-initialized data area 22
Program area 22
Section name 9, 22
Section initialization 70, 76
short 26, 147
show (option) 9
SH series 3
Sign 157
Sign extension 30
source (suboption) 9
Source listing information (C compiler listing) 12, 13
Stack area 24, 63
Higher addresses 34
Lower address 34
Stack frame 34, 168
frame size 15, 168
Stack pointer (SP) 34, 66, 68
Stack switching specification (interrupt function) 43, 44
Standard include file 4
Standard library file 4, 59
start (linkage editor option) 62
Static area size calculation 58
statistics (suboption) 9
Statistics information (Compiler listings) 12, 16
Status register (SR) 44, 47
Structure of object programs 24
Structure type 27, 150
Suboption 9
Systems 5
System installation 57
Initialization 69
Program configuration 67
Section initialization 70
Vector table setting 68

T
TRAPA instruction (interrupt function) 43
Trap-instruction return specification (interrupt function) 43, 44
Troubleshooting 54

U
Underflow 162
Union type 27, 150
unsigned 26, 147

V
Vector base register (VBR) 47
Vector table setting 67, 68, 74

W
width (suboption) 9
write routine (low-level interface routine) 86

X

Y

Z
Zero extension 28

HITACHI, LTD. SEMICONDUCTOR AND INTEGRATED CIRCUITS DIVISION SALES OFFICE

HEAD QUARTERS
Semiconductor & IC Div.
Karukozaka MN Bldg., 2-1, Ageba-cho,
Shinjuku-ku, Tokyo 162, Japan
Tel: Tokyo (03) 3266-9376
Fax: (03) 3235-2548

USA
Headquarters
Hitachi America, Ltd.
Semiconductor & IC Div.
2000 Sierra Point Parkway
Brisbane, CA. 94005-1819
Tel: 415-589-8300
Fax: 415-583-4207

Northwest Regional Office
1900 McCarthy Boulevard, Suite 310
Milpitas, CA. 95035
Tel: 408-954-8100
Fax: 408-954-0499

Southwest Regional Office
2030 Main St., Suite 450
Irvine, CA. 92714
Tel: 714-553-8500
Fax: 714-553-8561

South Central Regional Office
2 Lincoln Centre, Suite 865
5420 LBJ Freeway
Dallas, TX. 75240
Tel: 214-991-4510
Fax: 214-991-6151

Mid-Atlantic Regional Office
325 Columbia Turnpike, #203
Florham Park, NJ. 07932
Tel: 201-514-2100
Fax: 201-514-2020

North Central Regional Office
500 Park Boulevard, Suite 415
Itasca, IL. 60143
Tel: 312-773-4864
Fax: 312-773-9006

Northeast Regional Office
77 South Bedford St.
Burlington, MA. 01803
Tel: 617-229-2150
Fax: 617-229-6554

Automotive Regional Office
330 Town Centre Drive, Suite 311
Dearborn, MI. 48126
Tel: 313-271-4410
Fax: 313-271-5707

Pacific Mountain Region
4600 South Ulster St., Suite 700
Denver, CO 80237
Tel: 303-740-6644
Fax: 303-740-6609

Southeast Region
5511 Capital Center Dr., Suite 204
Raleigh, NC 27608
Tel: 919-233-0800
Fax: 919-233-0508

CANADA
Hitachi (Canadian) Ltd.
320 March Road, Suite 602
Kanata, Ont. K2K2E3 CANADA
Tel: 613-591-1990
Fax: 613-591-1994

EUROPE (CE)
Headquarters
Hitachi Europe GmbH
Electronic Components Div.
Continental Europe
Hans-Pinsel-Straße 10A
W-8013 Haar bei München, F. R. Germany
Tel: 089-46140
Fax: 089-463151

Sales Office
Hitachi Europe GmbH
Electronic Components Div.
North Germany/Benelux
Am Seestern 18; Postfach 11 05 36
W-4000 Düsseldorf 11
Tel: 0211-52830
Fax: 0211-5283779

Hitachi Europe GmbH
Electronic Components Div.
Central Germany
Fabrikstr. 17, W-7024 Filderstadt
Tel: 0711-772011
Fax: 0711-7775116

Hitachi Europe GmbH
Electronic Components Div.
South Germany/Austria
Hans-Pinsel-Str. 10a
W-8013 Haar bei München
Tel: 089-46140
Fax: 089-4614251

Hitachi Europe GmbH
Electronic Components Div.
Italy
Via L. Rizzo 8, 1-20151 Milano
Tel: 02-33404180
Fax: 02-33404152

Via Pescosolido 154, 1-00158 Roma
Tel: 06-4182151
Fax: 06-4182262

Hitachi Europe GmbH
Electronic Components Div.
Spain
c/Buganvilla, 5, E-28036 Madrid
Tel: 0034-17672782
Fax: 0034-13838511

Hitachi Europe GmbH
Electronic Components Div.
France
Immeuble Le Thales, 27,Rue Olof Palme
F-94006 Créteil
Tel: 1-45132345
Fax: 1-43398493

EUROPE (NE)
Headquarters

Hitachi Europe Ltd.
Electronic Components Div.
Northern Europe Headquarters
Whitebrook Park
Lower Cookham Road
Maidenhead
Berkshire SL6 8YA, United Kingdom
Tel: 0628-585000
Fax: 0628-778322

Branch Office

Hitachi Europe Ltd.
Electronic Components Div.
Northern Europe Headquarters
Haukadalsgatan 10
Box 1062, S-164 21 Kista, Sweden
Tel: 08-751-0035
Fax: 08-751-5073

ASIA
Headquarters

Hitachi Asia Pte. Ltd.
78 Shenton Way #11-01
Singapore 0207
Tel: 221-6131, 7355
Fax: 225-4225, 221-4474

Branch Office

Hitachi Asia Pte. Ltd.
Taipei Branch Office
9 th Fl. –1 No.64, Tun–Hwa N. Road
Taipei Financial Center
Taipei, Taiwan
Tel: 02-741-4021 to 6
Fax: 02-752-1567

ASIA(HK)
Headquarters

Hitachi Asia (Hong Kong) Ltd.
Unit 706, North Tower,
World Finance Centre, Harbour City
Canton Road, Tsimshatsui, Kowloon
Hong Kong
Tel: 7359218
Fax: 7306071

Branch Office

Hitachi Asia (Hong Kong) Ltd.
Seoul Branch Office
18 Floor Kukje Center Building
191, 2-Ka, Hanggang–Ro
Yongsan-Ku, Seoul, Korea
Tel: 796-3115, 3647 to 8
Fax: 796-2145

Hitachi Asia (Hong Kong) Ltd.
Beijing Office
Room 1412, Beijing Fortune Building,
5 Dong San Huan Bei-lu,
Chaoyang District, Beijing
People's Republic of China
Tel: 501-4351-4
Fax: 501-4350

SH Series C Compiler User's Manual

Publication Date: 1st Edition, June 1993
Published by: Semiconductor and IC Div.
Hitachi, Ltd.
Edited by: Technical Document Center
Hitachi Microcomputer System Ltd.
Copyright © Hitachi, Ltd., 1993. All rights reserved. Printed in Japan.

Hitachi Microcomputer

Support Software

SH Series
Cross Assembler
User's Manual

HITACHI

ADE-702-090A

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

HITACHI

Preface

This manual describes the SH Series Cross Assembler.

This manual is organized as follows:

Overview:	Gives an overview of the functions of the assembler.
Programmer's Guide:	Describes the assembler language syntax and programming techniques.
User's Guide:	Describes the use (invocation) of the assembler program itself.
Appendix:	Describes assembler limitations and error messages.

The related manuals are listed below.

For information concerning the SH microprocessor hardware:

"SH7032, SH7034 Hardware Manual"

For information concerning the SH microprocessor executable instructions:

"SH7000 Series Programming Manual"

For information concerning software development support tools:

"SH Series C Compiler User's Manual"
"H Series Linkage Editor User's Manual"
"H Series Librarian User's Manual"
"SH Series Simulator/Debugger User's Manual"

HITACHI

Notes:

- The following symbols have special meanings in this manual.

  <item>: <specification item>
  Δ: Blank space(s) or tab(s)
  >: The OS prompt (indicates the input waiting state)
  (RET): Press the Return (Enter) key.
  ... : The preceding item can be repeated.
  [ ]: The enclosed item is optional (i.e., can be omitted.)

- Numbers are written as follows in this manual.

  Binary: A prefix of "B'" is used.
  Octal: A prefix of "Q'" is used.
  Decimal: A prefix of "D'" is used.
  Hexadecimal: A prefix of "H'" is used.

  However, when there is no specification, the number without a prefix is decimal.

UNIX is an operating system administrated by the UNIX System Laboratories (United States).

MS-DOS is an operating system administrated by the Microsoft Corporation (United States).

Contents

Overview

Section 1 Overview 3

Section 2 Relationships between the Software Development Support Tools 5

Programmer's Guide

Section 1 Program Elements 11

1.1 Source Statements 11

1.1.1 Source Statement Structure 11

1.1.2 Coding of Source Statements 13

1.1.3 Coding of Source Statements across Multiple Lines 15

1.2 Reserved Words 16

1.3 Symbols 16

1.3.1 Functions of Symbols 16

1.3.2 Coding of Symbols 18

1.4 Constants 19

1.4.1 Integer Constants 19

1.4.2 Character Constants 20

1.5 Location Counter 21

1.6 Expressions 23

1.6.1 Elements of Expression 23

1.6.2 Operation Priority 25

1.6.3 Notes on Expressions 28

1.7 Character Strings 29

Section 2 Basic Programming Knowledge 31

2.1 Sections 31

2.1.1 Section Types by Usage 31

2.1.2 Absolute Address Sections and Relative Address Sections 36

2.2 Absolute and Relative Values 38

2.2.1 Absolute Values 38

2.2.2 Relative Values 38

2.3 Symbol Definition and Reference 39

2.3.1 Symbol Definition 39

2.3.2 Symbol Reference 41

HITACHI

2.4 Separate Assembly 43
2.4.1 Separate Assembly 43
2.4.2 Declaration of Export Symbols and Import Symbols 44

Section 3 Executable Instructions 47
3.1 Overview of Executable Instructions 47
3.2 Notes on Executable Instructions 52
3.2.1 Notes on the Operation Size 52
3.2.2 Notes on Delayed Branch Instructions 58
3.2.3 Notes on Address Calculations 59

Section 4 Assembler Directives 63
4.1 Overview of the Assembler Directives 63
4.2 Assembler Directive Reference 65
4.2.1 Section and Location Counter Assembler Directives 65
.SECTION ——— Section Declaration 66
.ORG ——— Location-Counter-Value Setting 71
.ALIGN ——— Location-Counter-Value Correction 73
4.2.2 Symbol Handling Assembler Directives 75
.EQU ——— Symbol Value Setting (resetting not allowed) 76
.ASSIGN ——— Symbol Value Setting (resetting allowed) 78
.REG ——— Alias of a Register Name Definition 80
4.2.3 Data and Data Area Reservation Assembler Directives 82
.DATA ——— Integer Data Reservation 83
.DATAB ——— Integer Data Block Reservation 85
.SDATA ——— Character String Data Reservation 88
.SDATAB ——— Character String Data Blocks Reservation 90
.SDATAC ——— Character String Data Reservation (with length) 93
.SDATAZ ——— Character String Data Reservation (with zero terminator) 95
.RES ——— Data Area Reservation 97
.SRES ——— Character String Data Area Reservation 100
.SRESC ——— Character String Data Area Reservation (with length) 102
.SRESZ ——— Character String Data Area Reservation (with zero terminator) 104
4.2.4 Export and Import Assembler Directives 106
.EXPORT ——— Export Symbols Declaration 107
.IMPORT ——— Import Symbols Declaration 109
.GLOBAL ——— Export and Import Symbols Declaration 111

4.2.5 Object Module Assembler Directives 113
.OUTPUT ——— Object Module Output Control 114
.DEBUG ——— Symbolic Debug Information Output Control 117
4.2.6 Assemble Listing Assembler Directives 119
.PRINT ——— Assemble Listing Output Control 120
.LIST ——— Source Program Listing Output Control 122
.FORM ——— Assemble Listing Line Count and Column Count Setting 126
.HEADING ——— Source Program Listing Header Setting 128
.PAGE ——— Source Program Listing New Page Insertion 130
.SPACE ——— Source Program Listing Blank Line Output 132
4.2.7 Other Assembler Directives 134
.PROGRAM ——— Object Module Name Setting 135
.RADIX ——— Default Integer Constant Radix Setting 137
.END ——— Source Program End Declaration 139

Section 5 File Inclusion Function 141
.INCLUDE ——— File Inclusion 142

Section 6 Conditional Assembly Function 145
6.1 Overview of the Conditional Assembly Function 145
6.1.1 Preprocessor Variables 145
6.1.2 Conditional Assembly 146
6.1.3 Iterated Expansion 147
6.1.4 Conditional Iterated Expansion 148
6.2 Conditional Assembly Directives 149
.ASSIGNA ——— Integer Preprocessor Variable Definition (redefinition is possible)..... 150
.ASSIGNC ——— Character Preprocessor Variable Definition (redefinition is possible)..... 153
.AIF .AELSE .AENDI ——— Conditional Assembly 155
.AREPEAT .AENDR ——— Iterated Expansion 157
.AWHILE .AENDW ——— Conditional Iterated Expansion 159
.EXITM ——— Expansion Termination 161

Section 7 Macro Function 163
7.1 Overview of the Macro Function 163
7.2 Macro Function Directives 165
.MACRO .ENDM ——— Macro Definition 166
.EXITM ——— Expansion Termination 169
7.3 Macro Body 171

7.4 Macro Call 175
7.5 Character String Manipulation Functions 177
.LEN ——— Character String Length Count 178
.INSTR ——— Character String Search 180
.SUBSTR ——— Character Substring Extraction 182

Section 8 Automatic Literal Pool Generation Function 185
8.1 Overview of Automatic Literal Pool Generation 185
8.2 Extended Instructions Related to Automatic Literal Pool Generation 186
8.3 Literal Pool Output 186
8.3.1 Literal Pool Output after Unconditional Branch (BRA, JMP, RTS, RTE) 187
8.3.2 Literal Pool Output to the .POOL Location 188
8.4 Literal Sharing 189
8.5 Literal Pool Output Suppression 190
8.6 Notes on Automatic Literal Pool Output 191

User's Guide

Section 1 Executing the Assembler 195
1.1 Command Line Format 195
1.2 File Specification Format 196

Section 2 Command Line Options 197
2.1 Overview of the Command Line Options 197
2.2 Command Line Option Reference 198
2.2.1 Object Module Command Line Options 198
OBJECT NOOBJECT ——— Object Module Output Control 199
DEBUG NODEBUG ——— Debug Information Output Control 201
2.2.2 Assemble Listing Command Line Options 203
LIST NOLIST ——— Assemble Listing Output Control 204
SOURCE NOSOURCE ——— Source Program Listing Output Control 206
CROSS_REFERENCE ——— Cross-Reference Listing Output Control ... 208
NOCROSS_REFERENCE
SECTION NOSECTION ——— Section Information Listing Output Control 210
SHOW NOSHOW ——— Source Program Listing Output Control 212
LINES ——— Sets the Number of Lines in the Assemble Listing 214
COLUMNS ——— Sets the Number of Columns in the Assemble Listing 215

Appendix

Appendix A Limitations and Notes on Programming 219

Appendix B Sample Program 220

Appendix C Assemble Listing Output Example 223
C.1 Source Program Listing 224
C.2 Cross-Reference Listing 225
C.3 Section Information Listing 226

Appendix D Error Messages 227
D.1 Error Types 227
D.2 Error Message Tables 229

Appendix E ASCII Code Table 241

Figures

Overview

Figure 1-1 Function of the Assembler 3
Figure 2-1 Relationships between the Software Development Support Tools 6

Programmer's Guide

Figure 2-1 Memory Reservation of Common Section 33
Figure 2-2 Data Structure Example Using Dummy Section 35
Figure 2-3 Meaning of the Terms Forward and Backward 41
Figure 2-4 Meaning of the Term External 41
Figure 2-5 Relationship between the Changed Range of the Program and the Range of the Program that must be Reassembled 43
Figure 3-1 Address Calculation Example (normal case) 59
Figure 3-2 Address Calculation Example (when the value of PC differs due to a branch) 60
Figure 3-3 Address Calculation Example (when SH microprocessor corrects the value of PC) 61
Figure 3-4 Address Calculation Example (when SH microprocessor does not correct the value of PC) 61

Appendix

Figure C-1 Source Program Listing Output Example 224
Figure C-2 Cross-Reference Listing Output Example 225
Figure C-3 Section Information Listing Output Example 226

Tables

Programmer's Guide

Table 1-1 Operators 23
Table 1-2 Operator Priority and Association Rules 25
Table 3-1 Addressing Modes 48
Table 3-2 Allowed Displacement Values 50
Table 3-3 Allowed Immediate Values 51
Table 3-4 Executable Instruction and Operation Size Combinations 52
Table 3-5 Relationship between Delayed Branch Instruction and Delay Slot Instructions 58
Table 4-1 Assembler Directives 63
Table 8-1 Extended Instructions and Expanded Results 186

User's Guide

Table 2-1 Command Line Options 197

Appendix

Table A-1 Limitations and Notes on Programming 219
Table D-1 Command Error Messages 229
Table D-2 Source Program Error Messages 230
Table D-3 Fatal Error Messages 240
Table E-1 ASCII Code Table 241

HITACHI

Overview

HITACHI

HITACHI

Section 1 Overview

The "SH Series Cross Assembler" (referred to below as the (or this) assembler) converts source programs written in assembly-language into a format that can be handled by SH microprocessors, and outputs the result as an object module. Also, the results of the assembly processing are output as an assemble listing.

This assembler provides the following functions to support efficient program development:

- Assembler directives
  Give the assembler various instructions.
- File inclusion function
  Includes files into a source file.
- Conditional assembly function
  Selects source statements to be assembled or repeats assembly according to a specified condition.
- Macro function
  Gives a name to a sequence of statements and defines it as one instruction.
- Automatic literal pool generation function
  Interprets data transfer instructions MOV.W #imm, MOV.L #imm, and MOVA #imm that are not provided by the SH microprocessor as extended instructions and expands them into SH microprocessor executable instructions and constant data (literals).

Figure 1-1 shows the function of the assembler.

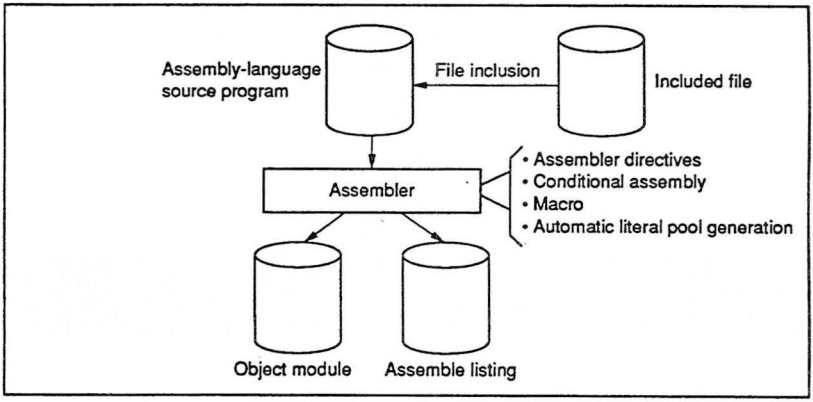


Figure 1-1 Function of the Assembler

HITACHI

Section 2 Relationships between the Software Development Support Tools

The following software development support tools are available for the SH microprocessors.

- SH Series C Compiler (Referred to below as the C compiler.)
- H Series Linkage Editor (Referred to below as the linkage editor.)
- H Series Librarian (Referred to below as the librarian.)
- H Series Object Converter (Referred to below as the object converter.)
- SH Series Simulator/Debugger (Referred to below as the simulator/debugger.)

Note: The linkage editor refers to version 5.0 or later.

These tools assist in the efficient development of application software.

Figure 2-1 shows the relationships between the software development support tools.

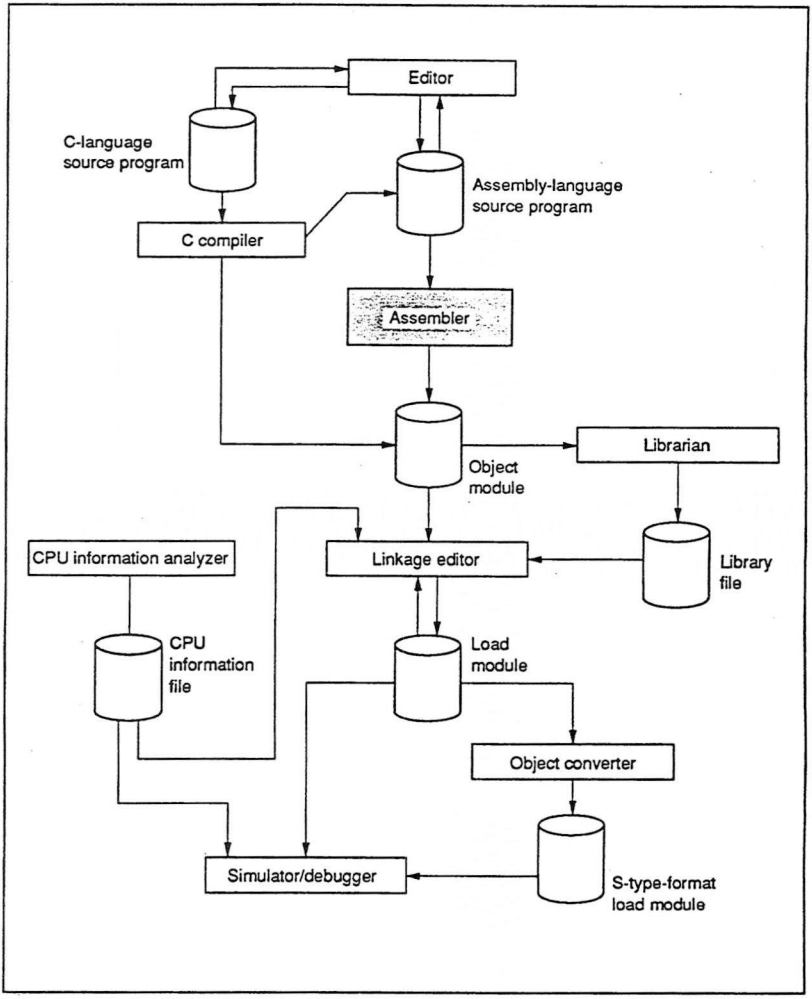


Figure 2-1 Relationships between the Software Development Support Tools

Supplement:

Use a general purpose editor (a text editor) to edit source programs.

The C compiler converts programs written in the C-language into either object modules or assembly-language source programs.

The librarian converts object modules and relocatable load modules into library files. We recommend handling processing that is common to multiple programs as a library file. (This has several advantages, including allowing modules to be easily managed.)

The linkage editor links together object modules and library files to produce load modules. (Load modules are programs in a format that a computer can execute.)

The object converter converts load modules into the S-type format. (The S-type format is a standard load module format.)

The simulator/debugger assists debugging microprocessor software.

Load modules created by this development support system can be input to several types of emulator. (Emulators are systems for debugging microprocessor system hardware and software.) Also, S-type-format load modules can be input into most EPROM programmers.

HITACHI

Programmer's Guide

HITACHI

HITACHI

Section 1 Program Elements

If source programs are compared to natural language writing, a source statement will correspond to "a sentence." The "words" that make up a source statement are reserved words and symbols. This section describes these basic program elements.

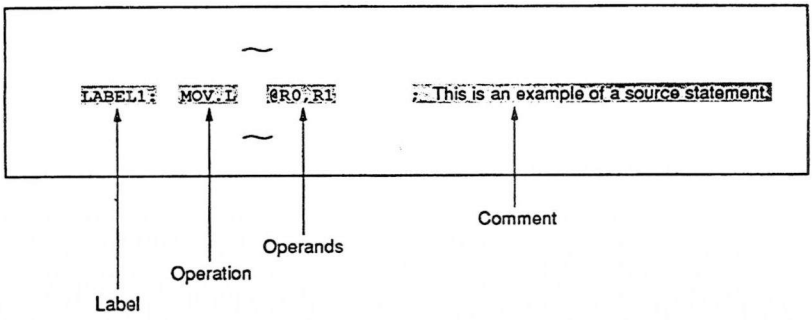
1.1 Source Statements

1.1.1 Source Statement Structure

The figure below shows the structure of a source statement.

```
[<label>] [Δ<operation>[Δ<operand(s)>]] [<comment>]
```

Example:



(1) **Label**

A symbol that is a tag attached to a source statement is written as a label.

A symbol is a name defined by the programmer.

(2) **Operation**

The mnemonic of an executable instruction, an extended instruction, an assembler directive, or a directive statement is written as the operation.

Executable instructions must be SH microprocessor instructions.

Extended instructions are instructions that are expanded into executable instructions and constant data (literals). For details, refer to Programmer's Guide, 8, "Automatic Literal Pool Generation Function".

Assembler directives are instructions that give directions to the assembler.

Directive statements are used for file inclusion, conditional assembly, and macro functions. For details on each of these functions, refer to Programmer's Guide, 5, "File Inclusion Function", 6, "Conditional Assembly Function", or 7, "Macro Function".

(3) **Operand**

The object(s) of the operation's execution are written as the operand.

The number of operands and their types are determined by the operation. There are also operations which do not require any operands.

(4) **Comment**

Notes or explanations that make the program easier to understand are written as the comment.

1.1.2 Coding of Source Statements

Source statements are written using ASCII characters.

In principle, a single statement must be written on a single line. The maximum length of a line is 255 bytes.

(1) Coding of Label

The label is written as follows:

- Written starting in the first column,
  Or:
- Written with a colon (:) appended to the end of the label.

Examples:

```
LABEL1                  ; This label is written starting in the first column.
 LABEL2:                ; This label is terminated with a colon.

-------------------------------------------------------------------------

     LABEL3             ; This label is regarded as an error by the assembler,
                        ; since it is neither written starting in the first column
                        ; nor terminated with a colon.
```

(2) Coding of Operation

The operation is written as follows:

- When there is no label:
  Written starting in the second or later column.

- When there is a label:
  Written after the label, separated by one or more spaces or tabs.

Examples:

```
        ADD     R0,R1       ; An example with no label.
LABEL1: ADD     R1,R2       ; An example with a label.
```

CAUTION!

Since white spaces and tabs are ASCII characters, each space or tab requires a byte of storage.

(3) Coding of Operand

The operand is written following the operation field, separated by one or more spaces or tabs.

Examples:

```
ADD     R0,R1       ; The ADD instruction takes 2 arguments.
SHAL    R1          ; The SHAL instruction takes 1 argument.
```

(4) Coding of Comment

The comment is written following a semicolon (;).

The assembler regards all characters from the semicolon to the end of the line as the comment.

Examples:

```
ADD     R0,R1       ; Adds R0 to R1.
```

1.1.3 Coding of Source Statements across Multiple Lines

We recommend writing a single source statement across several lines in the following situations:

- When the source statement is too long as a single statement.
- When it is desirable to attach a comment to each operand.

Write source statements across multiple lines using the following procedure.

(a) Insert a new line writing a comma that separates operands as the point to break the line.
(b) Insert a plus sign (+) in the first column of the next line.
(c) Continue writing the source statement following the plus sign.

Spaces and tabs can be inserted following the plus sign.

Examples:

```
        .DATA.L   H'FFFF0000,
                  H'FF00FF00,
                  H'FFFFFFFF

; In this example, a single source statement is written across three lines.
```

A comment can be attached at the end of each line.

Examples:

```
        .DATA.L   H'FFFF0000,     ; Initial value 1.
+                 H'FF00FF00,     ; Initial value 2.
+                 H'FFFFFFFF      ; Initial value 3.

; This is an example of attaching a comment to each operand.
```

1.2 Reserved Words

Reserved words are names that the assembler reserves as symbols with special meanings.

This assembler uses the following reserved words.

- Register names

R0	R1	R2	R3	R4	R5
R6	R7	R8	R9	R10	R11
R12	R13	R14	R15	SP*	
SR	GBR	VBR	MACH	MACL	PR
PC					

  Note: * R15 and SP indicate the same register.

- Operators (STARTOF, SIZEOF)
- The location counter symbol, the dollar sign (S: a single character symbol)

Reserved words cannot be used as user-defined symbols.

Reference:

Operators → Programmer's Guide, 1.6.1, "Expression Elements"
Location counter → Programmer's Guide, 1.5, "Location Counter"
Symbols → Programmer's Guide, 1.3, "Symbols"

1.3 Symbols

1.3.1 Functions of Symbols

Symbols are names defined by the programmer, and perform the following functions.

- Address symbols ..... Express data storage and destination addresses.
- Constant symbols ..... Express constants.
- Aliases of register name ..... Express general registers.
- Section names..... Express section names. *.

Note: * A section is a part of program, and the linkage editor regards it as a unit of processing.

The following shows examples of symbol usages.

Examples:

```
                ~
        BRA     SUB1            ; BRA is a branch instruction.
                                ; SUB1 is the address of the destination.
                ~
SUB1:
                ~
-------------------------------------------------------------------------
                ~
MAX:    .EQU    100             ; .EQU is an assembler directive that sets a value to a
                                ; symbol.
        MOV     #MAX,R0         ; MAX expresses the constant value 100.
                ~
-------------------------------------------------------------------------
                ~
MIN:    .REG    (R0)            ; .REG is an assembler directive that defines a register
                                ; name.
        MOV     #100,MIN        ; Here, MIN is a name for R0.
                ~
-------------------------------------------------------------------------
                ~
        .SECTION CD,CODE,ALIGN=4
                                ; .SECTION is an assembler directive that declares a section.
                                ; CD is the name of the current section.
                ~
```

1.3.2 Coding of Symbols

(1) Available Characters

The following members of the ASCII character can be used.

- Upper-case and lower-case letters (A to Z, a to z)
- Numbers (0 to 9)
- The underscore character (_)
- The dollar sign character ($)

The assembler distinguishes upper-case and lower-case in symbols.

(2) First Character in a Symbol

The first character in a symbol must be one of the following.

- Upper-case and lower-case letters (A to Z, a to z)
- The underscore character (_)
- The dollar sign character ($)

(3) Maximum Length of a Symbol

A symbol may contain up to 32 characters.

The assembler ignores any characters after the first 32.

(4) Names that Cannot Be Used as Symbols

Reserved words cannot be used as symbols. The following names must not be used because they are used as internal symbols by the assembler.

_$$nnnnn (n is a number from 0 to 9.)

Note: Internal symbols are necessary for assembler internal processing. Internal symbols are not output to assemble listings or object modules.

CAUTION!

The dollar sign character used alone is a reserved word that expresses the location counter.

References:

Reserved words → Programmer's Guide, 1.2, "Reserved Words"

1.4 Constants

1.4.1 Integer Constants

Integer constants are expressed with a prefix that indicates the radix.

The radix indicator prefix is a notation that indicates the base of the constant.

- Binary numbers ..... The radix indicator "B'" plus a binary constant.
- Octal numbers ..... The radix indicator "Q'" plus an octal constant.
- Decimal numbers ..... The radix indicator "D'" plus a decimal constant.
- Hexadecimal numbers ..... The radix indicator "H'" plus a hexadecimal constant.

The assembler does not distinguish upper-case and lower-case letters in the radix indicator.

The radix indicator and the constant value must be written with no intervening space.

Examples:

```
.DATA.B  B'10001000  ;
.DATA.B  Q'210       ; These source statements all the same
.DATA.B  D'136       ; numerical value.
.DATA.B  H'88        ;
```

The radix indicator can be omitted. Integer constants with no radix indicator are normally decimal constants, although the radix for such constants can be changed with the .RADIX assembler directive.

References:

Interpretation of integer constants without a radix specified
→ Programmer's Guide, 4.2.7, "Other Assembler Directives", .RADIX

Supplement:

"Q" is used instead of "O" to avoid confusion with the digit 0.

1.4.2 Character Constants

Character constants are considered to be constants that represent ASCII codes.

Character constants are written by enclosing up to 4 ASCII characters in double quotation marks.

The following ASCII characters can be used in character constants.

ASCII codes { H'09 (tab)
H'20 (space) to H'7E (tilde)

Examples:

```
.DATA.L  "ABC"     ; This is the same as .DATA.L H'00414243.
.DATA.W  "AB"      ; This is the same as .DATA.W H'4142.
.DATA.B  "A"       ; This is the same as .DATA.B H'41.

                   ; The ASCII code for A is: H'41
                   ; The ASCII code for B is: H'42
                   ; The ASCII code for C is: H'43
```

Use two double quotation marks in succession to indicate a single double quotation mark in a character constant.

Example:

```
.DATA.B  """"      ; This is a character constant consisting of a single
                     double quotation mark.
```

1.5 Location Counter

The location counter expresses the address (location) in memory where the corresponding object code (the result of converting executable instructions and data into codes the microprocessor can regard) is stored.

The value of the location counter is automatically adjusted according to the object code output.

The value of the location counter can be changed intentionally using assembler directives.

Examples:

```
.ORG     H'00001000      ; This assembler directive sets the location counter to
                         ; H'00001000.

.DATA.W  H'FF            ; The object code generated by this assembler directive has
                         ; a length of 2 bytes.
                         ; The location counter changes to H'00001002.

.DATA.W  H'F0            ; The object code generated by this assembler directive has
                         ; a length of 2 bytes.
                         ; The location counter changes to H'00001004.

.DATA.W  H'10            ; The object code generated by this assembler directive has
                         ; a length of 2 bytes.
                         ; The location counter changes to H'00001006.

.ALIGN   4               ; The value of the location counter is corrected to be a multiple
                         ; of 4.
                         ; The location counter changes to H'00001008.

.DATA.L  H'FFFFFFFF      ; The object code generated by this assembler directive has
                         ; a length of 4 bytes.
                         ; The location counter changes to H'0000100C.

;        .ORG is an assembler directive that sets the value of the location counter.
;        .ALIGN is an assembler directive that adjusts the value of the location counter.

;        .DATA is an assembler directive that reserves data in memory.
;        .W is a specifier that indicates that data is handled in word (2 byte) size.
;        .L is a specifier that indicates that data is handled in long word (4 byte) size.
```

References:

Setting the value of the location counter
→ Programmer's Guide, 4.2.1, "Section and Location Counter Assembler Directives" .ORG

Correcting the value of the location counter
→ Programmer's Guide, 4.2.1, "Section and Location Counter Assembler Directives" .ALIGN

The location counter is referenced using the dollar sign symbol.

Examples:

```
LABEL1:  .EQU    $                ; This assembler directive sets the value of the
                                  ; location counter to the symbol LABEL1.

;        .EQU is an assembler directive that sets the value to a symbol.
```

1.6 Expressions

Expressions are combinations of constants, symbols, and operators that derive a value, and are used as the operands of executable instructions and assembler directives.

1.6.1 Elements of Expression

An expression consists of terms, operators, and parentheses.

(1) Terms

The terms are the followings:

- A constant
- The location counter reference ($)
- A symbol (excluding aliases of the register name)
- The result of a calculation specified by a combination of the above terms and an operator.

An independent term is also a type of expression.

(2) Operators

Table 1-1 shows the operators supported by the assembler.

Table 1-1 Operators

Operator Type	Operator	Operation	Coding
Arithmetic operations	+	Unary plus	+ <term>
	–	Unary minus	– <term>
	+	Addition	<term1> + <term2>
	–	Subtraction	<term1> – <term2>
	*	Multiplication	<term1> * <term2>
	/	Division	<term1> / <term2>
Logic operations	~	Unary negation	~ <term>
	&	Logical AND	<term1> & <term2>
	|	Logical OR	<term1> | <term2>
	~	Exclusive OR	<term1> ~ <term2>
Section set operations*	STARTOF	Derives the starting address of a section set.	STARTOF <section name>
	SIZEOF	Derives the size in bytes of a section set.	SIZEOF <section name>

Note: * See the supplement on the following page.

(3) Parentheses

Parentheses modify the operation priority.

See the next section, section 1.6.2, "Operation Order", for a description of the use of parentheses.

Supplement:

In this assembly-language, programs are divided into units called section. Sections are the units in which linkage processing is performed.

When there are multiple sections of the same type and same name within a given program, the linkage editor links them into a single "section set".

STARTOF is an operator that determines the starting address of the section set.

SIZEOF is an operator that determines the size of the section set in byte units.

References:

Sections → Programmer's Guide, 2.1, "Sections"

1.6.2 Operation Priority

When multiple operations appear in a single expression, the order in which the processing is performed is determined by the operator priority and by the use of parentheses. The assembler processes operations according to the following rules.

<Rule 1>
Processing starts from operations enclosed in parentheses.
When there are multiple parentheses, processing starts with the operations surrounded by the innermost parentheses.

<Rule 2>
Processing starts with the operator with the highest priority.

<Rule 3>
Processing proceeds in the direction of the operator association rule when operators have the same priority.

Table 1-2 shows the operator priority and the association rule.

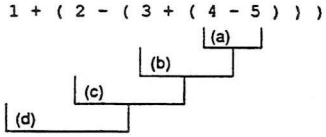
Table 1-2 Operator Priority and Association Rules

Priority	Operator	Association Rule
1 (high)	+ – ~ STARTOF SIZEOF*	Operators are processed from right to left.
2	* /	Operators are processed from left to right.
3	+ –	Operators are processed from left to right.
4	&	Operators are processed from left to right.
5 (low)	| ~	Operators are processed from left to right.

Note: * The operators of priority 1 (highest priority) are for unary operation.

The figures below show examples of expressions.

Example 1:

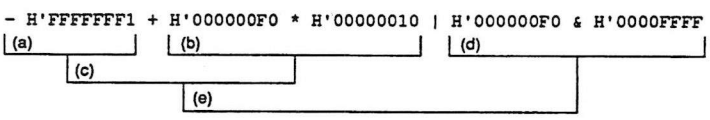


The assembler calculates this expression in the order (a) to (d).

The result of (a) is –1
The result of (b) is 2
The result of (c) is 0
The result of (d) is 1

} The final result of this calculation is 1.

Example 2:

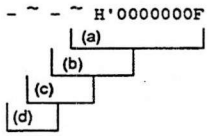


The assembler calculates this expression in the order (a) to (e).

The result of (a) is H'0000000F
The result of (b) is H'00000F00
The result of (c) is H'00000F0F
The result of (d) is H'000000F0
The result of (e) is H'00000FFF

} The final result of this calculation is H'00000FFF.

Example 3:



The assembler calculates this expression in the order (a) to (d).

The result of (a) is H'FFFFFFF0 -
The result of (b) is H'00000010
The result of (c) is H'FFFFFFEF
The result of (d) is H'00000011

The final result of this calculation is H'00000011.

1.6.3 Notes on Expressions

(1) Internal Processing

The assembler regards expression values as 32-bit signed values.

Example:

```
~H'F0
```

The assembler regards H'F0 as H'000000F0.
Therefore, the value of ~H'F0 is H'FFFFFF0F. (Note that this is not H'0000000F.)

(2) Arithmetic Operators

The multiplication and division operators cannot take terms that contain relative values (values which are not determined until the end of the linkage process) as their operands.

Also, a divisor of 0 cannot be used with the division operator.

(3) Logic Operators

The logic operators cannot take terms that contain relative values as their operands.

References:

Relative values → Programmer's Guide, 2.2, "Absolute and Relative Values".

1.7 Character Strings

Character strings are sequences of character data.

The following ASCII characters can be used in character strings.

ASCII codes { H'09 (tab)
H'20 (space) to H'7E (tilde)

A single character in a character string has as its value the ASCII code for that character and is represented as a byte sized data object.

Character strings are written enclosed in double quotation marks.

Use two double quotation marks in succession to indicate a single double quotation mark in a character string.

Examples:

```
        .SDATA   "Hello!"        ; This statement reserves the character string data
                                 ; Hello!
        .SDATA   """Hello!"""    ; This statement reserves the character string data
                                 ; "Hello!"

;       .SDATA is an assembler directive that reserves character string data in memory.
```

Supplement:

The difference between character constants and character strings is as follows.

Character constants are numeric values. They have a data size of either 1 byte, 2 bytes, or 4 bytes.

Character strings cannot be handled as numeric values. A character string has a data size of between 1 byte and 255 bytes.

HITACHI

Section 2 Basic Programming Knowledge

This section presents the basic knowledge required for programming in assembly-language.

2.1 Sections

If source programs are compared to natural language writing, a section will correspond to a "chapter." The section is the processing unit used when the linkage editor links object modules.

2.1.1 Section Types by Usage

Sections are classified by usage into the following types.

- Code section
- Data section
- Common section
- Stack section
- Dummy section

(1) Code Section

The following can be written in a code section:

- Executable instructions
- Extended instructions
- Assembler directives that reserve initialized data.

Examples:

```
        .SECTION   CD,CODE,ALIGN=4      ; This assembler directive declares a
                                        ; code section with the name CD.

        MOV.L      X,R1                 ; This is an executable instruction.
        MOV        R1,R2
          ~

        .ALIGN     4
X:      .DATA.L    H'FFFFFFFF           ; This assembler directive reserves
                                        ; initialized data.
          ~
```

(2) Data Section

The following can be written in a data section:

- Assembler directives that reserve initialized data.
- Assembler directives that reserve uninitialized data.

Examples:

```
        .SECTION    DT1,DATA,ALIGN=4        ; This assembler directive declares
                                            ; a data section with the name DT1.

        .DATA.W     H'FF00                  ; These assembler directives reserve
        .DATA.B     H'FF                    ; initialized data.

             ~

        .SECTION    DT2,DATA,ALIGN=4        ; This assembler directive declares
                                            ; a data section with the name DT2.

        .RES.W      10                      ; These assembler directives reserve
        .RES.B      10                      ; data areas that do not have initial
                                            ; values.

             ~
```

(3) Common Section

A common section is used as a section to hold data that is shared between files when a source program consists of multiple source files.

The following can be written in a common section:

- Assembler directives that reserve initialized data.
- Assembler directives that reserve uninitialized data.

Supplement:

The linkage editor reserves common sections with the same name to the same area in memory. In the example shown in figure 2-1, the common section CM declared in file A and the common section CM declared in file B are reserved to the same area in memory.

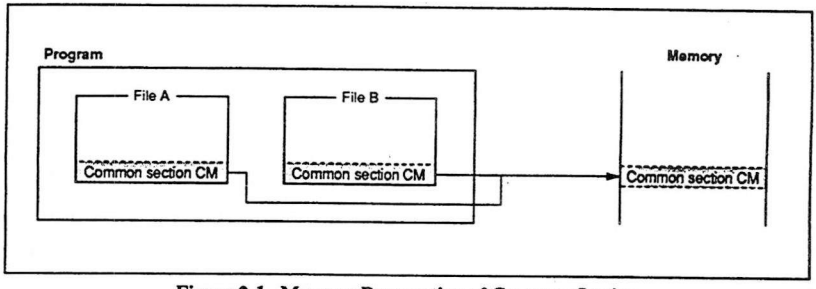


Figure 2-1 Memory Reservation of Common Section

(4) Stack Section

The section that SH microprocessors use as a stack area (an area for temporary data storage) is called the stack section.

The following can be written in the stack section:

- Assembler directives that reserve uninitialized data.

Examples:

```
        .SECTION ST,STACK,ALIGN=4       ; This assembler directive declares a stack
                                        ; section with the name ST.

        .RES.B   1024                   ; This assembler directive reserves a stack
                                        ; area of 1024 bytes.
```

(5) Dummy Section

A dummy section is a hypothetical section for representing data structures. The assembler does not output dummy sections to the object module.

The following can be written in a dummy section:

- Assembler directives that reserve uninitialized data.

Examples:

```
        .SECTION DM,DUMMY               ; This assembler directive declares
                                        ; a dummy section with the name DM.
        .RES.B  1                       ; The assembler does not output the
A:      .RES.B  1                       ; section DM to the object module.
B:      .RES.B  2
          ~
```

Specific methods for specifying data structures are described in the supplement on the next page.

Supplement:

As shown in figure 2-2, it is possible to access areas in memory by using address symbols from a dummy section.

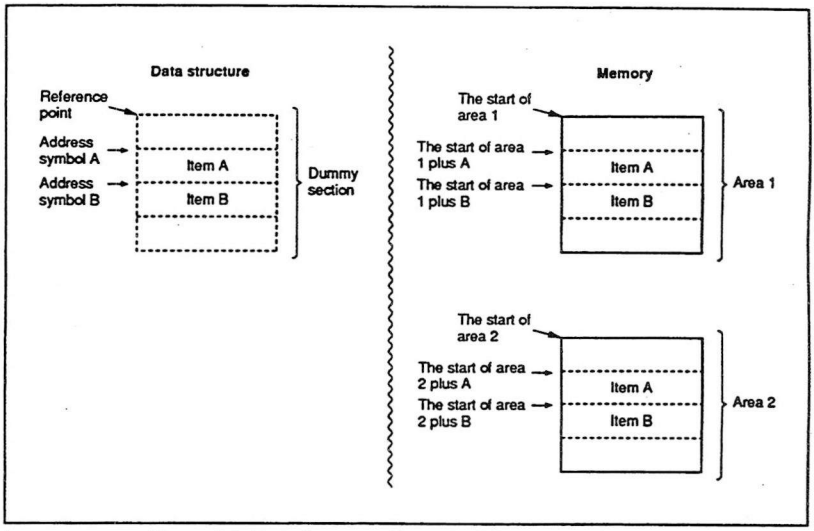


Figure 2-2 Data Structure Example Using Dummy Section

Coding Example:

```
        ~

; In the example above,
; assume that R1 holds the starting address of area 1 and R2 holds the starting address of
; area 2.

        MOV.L   @(B,R1),R0      ; Moves the contents of item B in area 1 to R0.
        MOV.L   R0,@(B,R2)      ; Moves the contents of R0 to item B in area 2.

        ~
```

CAUTION!

1. The following cannot be used in stack and dummy sections:
   - Executable instructions
   - Extended instructions
   - Assembler directives that reserve initialized data (.DATA, .DATAB, .SDATA, .SDATAB, .SDATAC, and .SDATAZ)
2. When using a data or common section, be sure to keep in mind whether that section is reserved to ROM or RAM.

2.1.2 Absolute Address Sections and Relative Address Sections

A section can be classified as either an absolute address section or as a relative address section depending on whether absolute start addresses are given to the sections at assembly.

(1) Absolute Address Sections

The memory location of absolute address sections is specified in the source program, and cannot be changed by the linkage editor. In this assembly language, locations in an absolute address section are expressed as absolute addresses, which are addresses that express the position in memory itself.

Examples:

```
.SECTION ABS,DATA,LOCATE=H'0000F000  ; ABS is an absolute-address section.
                                     ; The starting address of section ABS is
                                     ; the absolute address H'0000F000.

.DATA.W  H'1111                      ; The constant H'1111 is reserved at
                                     ; the absolute address H'0000F000.

.DATA.W  H'2222                      ; The constant H'2222 is reserved at
                                     ; the absolute address H'0000F002.
```

(2) Relative Address Section

The location in memory of relative sections is not specified in the source program, but rather is determined when the sections are linked by the linkage editor. In this assembly-language, locations in a relative address section are expressed as relative addresses, which are addresses that express the position relative to the start of the section itself.

Examples:

```
.SECTION  REL,DATA,ALIGN=4      ; REL is a relative address section.
                                ; The starting address of section REL is
                                ; determined after linkage.

.DATA.W   H'1111                ; The constant H'1111 is reserved at the
                                ; relative address H'00000000.

.DATA.W   H'2222                ; The constant H'2222 is reserved at the
                                ; relative address H'00000002.
```

Supplement:

Dummy sections correspond neither to relative nor to absolute sections.

2.2 Absolute and Relative Values

Absolute values are determined when assembly completes. Relative values are not determined until the linkage editor completes.

2.2.1 Absolute Values

The following are the absolute values handled by the assembler.

(1) Constants

- Integer constants
- Character constants
- Symbols that have a value that is one of the above (referred to below as constant symbols).

(2) Absolute Address Values

- The location counter referenced in an absolute address section
- The location counter referenced in a dummy section
- Symbols that have a value that is one of the above (referred to below as absolute address symbols).

(3) Other Absolute Values

Expressions whose value is determined when assembly completes.

2.2.2 Relative Values

The following are the relative values handled by the assembler.

(1) Relative Address Values

- The location counter referenced in a relative address segment
- Symbols that have the above as a value. (Such symbols are referred to as relative address symbols.)

(2) External Reference Values

Symbols that reference another file (referred to below as import symbols).

(3) Other Relative Values

Expressions whose value is not determined until the linkage editor completes.

2.3 Symbol Definition and Reference

2.3.1 Symbol Definition

(1) Normal Definition

The normal method for defining a symbol consists of writing that symbol in the label field of a source statement. The value of that symbol will then be the value of the location counter at that point in the program.

Examples:

```
        .SECTION DT1,DATA,LOCATE=H'0000F000      ; This statement declares an
                                                 ; absolute address section.
X1:     .DATA.W  H'1111                  ; The value of X1 becomes H'0000F000.
X2:     .DATA.W  H'2222                  ; The value of X2 becomes H'0000F002.
          ~
```

```
      .SECTION DT2,DATA,ALIGN=4          ; This statement declares a relative address
                                         ; section.
Y1:   .DATA.W  H'1111                    ; The value of Y1 is determined when the
                                         ; linkage editor completes, and its value is
                                         ; the start address of the section.
Y2:   .DATA.W  H'2222                    ; The value of Y2 is determined when the
                                         ; linkage editor completes, and its value is
                                           the start address of the section plus 2.
```

(2) Definition by Assembler Directive

Symbols can be defined by using assembler directives to set an arbitrary value or a special meaning.

Examples:

```
        .SECTION DT1,DATA,ALIGN=4   ; DT1 is the section name.
                                    ; A section name is also a type of symbol,
                                    ; a symbol that expresses the start
                                    ; address of a section.
                                    ; However, the syntactic handling of address
                                    ; symbols and section names is different.

X:      .EQU     100                ; The value of X is 100.
                                    ; X cannot be redefined.

Y:      .ASSIGN  10                 ; The value of Y is 10.
                                    ; Y can be redefined.

Z:      .REG     (R1)               ; Z becomes an alias of the general
                                    ; register R1.
                                    ; Z cannot be redefined.
```

2.3.2 Symbol Reference

There are three forms of symbol reference as follows:

- Forward reference
- Backward reference
- External reference

Supplement:

Figure 2-3 shows the meaning of the terms forward and backward as used in this manual.

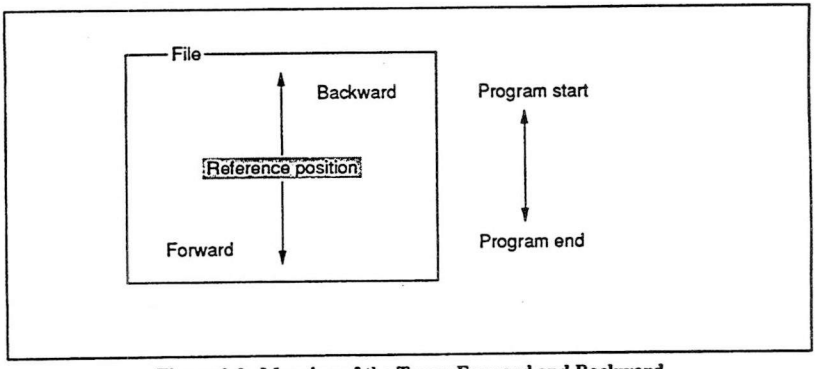


Figure 2-3 Meaning of the Terms Forward and Backward

Figure 2-4 shows the meaning of the term external as used in this manual.

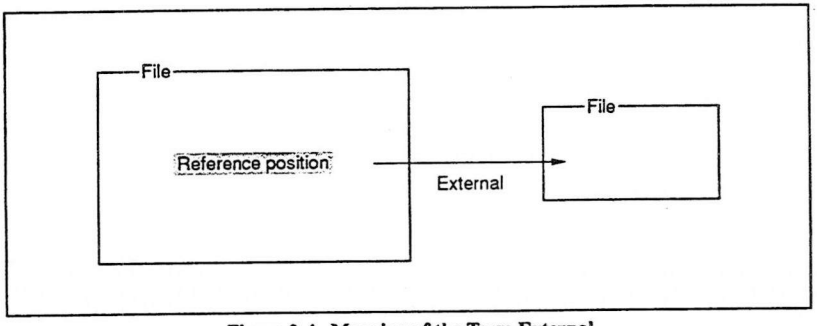


Figure 2-4 Meaning of the Term External

(1) Forward Reference

Forward reference means referencing a symbol that is defined forward from the point of reference, i.e., is defined later in the program.

Examples:

```
            ~
        BRA     FORWARD     ; BRA is a branch instruction.
                            ; This is a forward reference to the symbol FORWARD.
            ~
FORWARD:
            ~
```

(2) Backward Reference

Backward reference means referring to a symbol that is defined backward from the point of reference, i.e., is defined earlier in the program.

Examples:

```
            ~
BACK:
            ~
        BRA     BACK        ; BRA is a branch instruction.
                            ; This is a backward reference to the symbol BACK.
            ~
```

(3) External Reference

When a source program consists of multiple source files, a reference to a symbol defined in another file is called an external reference. External reference is described in the next section, 2.4, "Separate Assembly".

2.4 Separate Assembly

2.4.1 Separate Assembly

Separate assembly refers to the technique of creating a source program in multiple separate source files, and finally creating a single load module by linking together those source files' object modules using the linkage editor.

The process of developing software often consists of repeatedly correcting and reassembling the program. In such cases, if the source program is partitioned, it will be only necessary to reassemble the source file that was changed. As a result, the time required to construct the complete program will be significantly reduced.

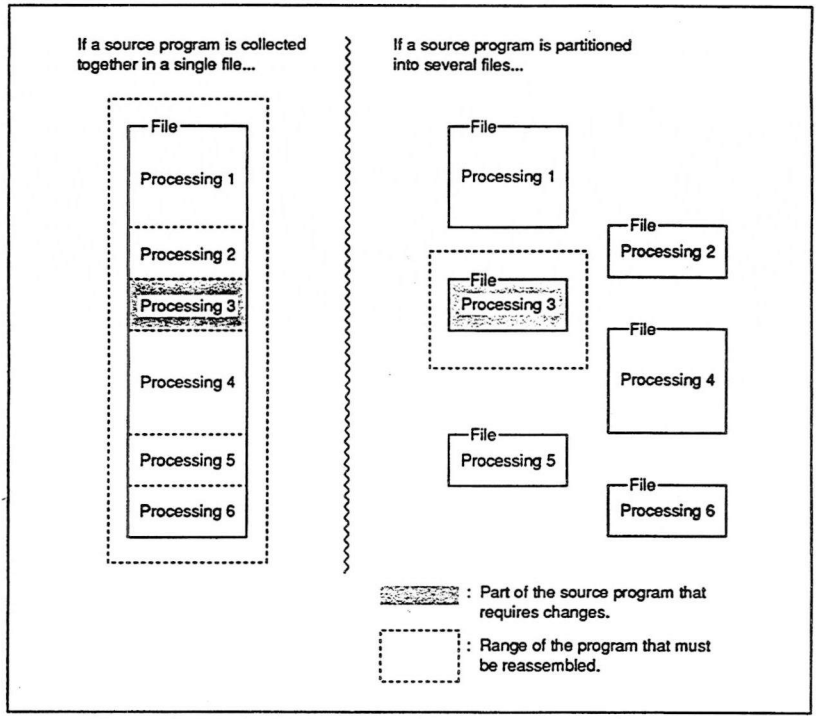


Figure 2-5 Relationship between the Changed Range of the Program and the Range of the Program that must be Reassembled

The procedure involved in separate assembly consists of steps (a) to (d).

(a) Investigate methods for partitioning the program.

Normally, programs are partitioned by function.

Note that the memory reservation of the section must also be considered at this point.

(b) Divide the source program into separate files and edit those files accordingly.

(c) Assemble the individual files.

(d) Link the individual object modules into a single load module.

2.4.2 Declaration of Export Symbols and Import Symbols

When a source program consists of multiple files, referencing a symbol defined in one file from another file is called "external reference" or "import." When referencing a symbol externally (this declaration is called "external definition" or "export"), it is necessary to declare to the assembler that "this symbol is shared between multiple files."

(1) Export Symbol Declaration

This declaration is used to declare that the definition of the symbol is valid in other files. .EXPORT or .GLOBAL assembler directive is used to make this declaration.

(2) Import Symbol Declaration

This declaration is used to declare that a symbol is defined in another file. .IMPORT or .GLOBAL assembler directive is used to make this declaration.

Examples:

In this example the symbol MAX is defined in file A and referenced in file B.

File A:

```
            ~
            ~
            .EXPORT  MAX          ; Declares MAX to be an export symbol.
MAX:        .EQU     100          ; Defines MAX.
            ~
```

File B:

```
            ~
            ~
            .IMPORT  MAX          ; Declares MAX to be an import symbol.
            MOV      #MAX,R0      ; References MAX.
            ~
```

References:

Symbol Export and Import

→ Programmer's Guide, 4.2.4, "Export and Import Assembler Directives", .EXPORT, .IMPORT, .GLOBAL

HITACHI

Section 3 Executable Instructions

This section describes the points that must be kept in mind when using executable instructions in this assembler.

3.1 Overview of Executable Instructions

The executable instructions are the instructions of SH microprocessor. SH microprocessor interprets and executes the executable instructions in the object code stored in memory.

An executable instruction source statement has the following basic form.

```
[<symbol>:] Δ<mnemonic>[.<operation size>] [Δ<addressing mode>[,<addressing mode]] [;<comment>]
 Label       Operation                      Operand                                  Comment
```

This section describes the mnemonic, operation size, and addressing mode. The other elements are described in detail in section 1, "Program Elements", in the Programmer's Guide.

(1) Mnemonic

The mnemonic expresses the type of executable instruction. Abbreviations that indicate the type of processing are provided as mnemonics for SH microprocessor instructions.

The assembler does not distinguish upper-case and lower-case letters in mnemonics.

(2) Operation Size

The operation size is the unit for processing data. The operation sizes vary with the executable instruction. The assembler does not distinguish upper-case and lower-case letters in the operation size.

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

(3) Addressing Mode

The addressing mode specifies the data area accessed, and the destination address. The addressing modes vary with the executable instruction. Table 3-1 shows the addressing mode.

Table 3-1 Addressing Modes

Addressing Mode	Name	Description
Rn	Register direct	The contents of the general register Rn.
SR	SR direct	The contents of the status register (SR).
GBR	GBR direct	The contents of the global base register (GBR).
VBR	VBR direct	The contents of the vector base register (VBR).
MACH	MACH direct	The contents of the accumulator register (MACH).
MACL	MACL direct	The contents of the accumulator register (MACL).
PR	PR direct	The contents of the procedure register (PR).
@Rn	Register indirect	A memory location. The value in Rn gives the start address of the memory accessed.
@Rn+	Register indirect with post-increment	A memory location. The value in Rn (before being incremented*1) gives the start address of the memory accessed. SH microprocessor first uses the value in Rn for the memory reference, and increments Rn afterwards.
@–Rn	Register indirect with pre-decrement	A memory location. The value in Rn (after being decremented*2) gives the start address of the memory accessed. SH microprocessor first decrements Rn, and then uses that value for the memory reference.
@(disp,Rn)	Register indirect with displacement*3	A memory location. The start address of the memory access is given by: the value of Rn plus the displacement (disp). The value of Rn is not changed.
@(R0,Rn)	Register indirect with index	A memory location. The start address of the memory access is given by: the value of R0 plus the value of Rn. The values of R0 and Rn are not changed.
@(disp,GBR)	GBR indirect with displacement	A memory location. The start address of the memory access is given by: the value of GBR plus the displacement (disp). The value of GBR is not changed.

Notes 1 to 3: See next page.

Table 3-1 Addressing Modes (cont)

Addressing Mode	Name	Description
@(R0,GBR)	GBR indirect with index	A memory location. The start address of the memory access is given by: the value of GBR plus the value of R0. The values of GBR and R0 are not changed.
@(disp,PC)	PC relative with displacement	A memory location. The start address of the memory access is given by: the value of the PC plus the displacement (disp).
symbol	PC relative specified with symbol	[When used as the operand of a branch instruction] The symbol directly indicates the destination address. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol – PC.
		[When used as the operand of a data move instruction] A memory location. The symbol expresses the starting address of the memory accessed. The assembler derives a displacement (disp) from the symbol and the value of the PC, using the formula: disp = symbol – PC.
#imm	Immediate	Expresses a constant.

Note: Rn ..... A general register (R0 to R15)*4
SR ..... The status register
VBR ..... The vector base register
PR ..... The procedure register
R0 ..... The general register R0 (when only R0 can be specified)
GBR ..... The global base register
MACH, MACL ..... The accumulator register
PC ..... The program counter

Notes: 1. Increment
The amount of the increment is 1 when the operation size is a byte, 2 when the operation size is a word, and 4 when the operation size is a long word.

2. Decrement
The amount of the decrement is 1 when the operation size is a byte, 2 when the operation size is a word, and 4 when the operation size is a long word.

3. Displacement
A displacement is the distance between two points. In this assembly-language, the unit of displacement values is in bytes.

4. Concerning R15
R15 can also be specified by the symbol SP. SP is an abbreviation for stack pointer, a pointer to the stack area.

The values that can be used for the displacement vary with the addressing mode and the operation size.

Table 3-2 Allowed Displacement Values

Addressing Mode	Displacement*
@(disp,Rn)	When the operation size is byte (B): H'00000000 to H'0000000F (0 to 15)
	When the operation size is word (W): H'00000000 to H'0000001F (0 to 31)
	When the operation size is long word (L): H'00000000 to H'0000003F (0 to 63)
@(disp,GBR)	When the operation size is byte (B): H'00000000 to H'000000FF (0 to 255)
	When the operation size is word (W): H'00000000 to H'000001FF (0 to 511)
	When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)
@(disp,PC)	When the operation size is word (W): H'00000000 to H'000001FF (0 to 511)
	When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)
symbol	[When used as a branch instruction operand]
	When used as an operand for a conditional branch instruction (BT or BF): { H'00000000 to H'000000FF (0 to 255) H'FFFFFF00 to H'FFFFFFFF (–256 to -1)
	When used as an operand for an unconditional branch instruction (BRA, BSR, JMP, JSR, or RTS) { H'00000000 to H'00000FFF (0 to 4095) H'FFFFF000 to H'FFFFFFFF (–4096 to -1)
	[When used as the operand of a data move instruction]
	When the operation size is word (W): H'00000000 to H'000001FF (0 to 511)
	When the operation size is long word (L): H'00000000 to H'000003FF (0 to 1023)

Note: * Units are bytes, numbers in parentheses are decimal.

The values that can be used for immediate values vary with the executable instruction.

Table 3-3 Allowed Immediate Values

Executable Instruction	Immediate Value	
TST, AND, OR, XOR	H'00000000 to H'000000FF	(0 to 255)
MOV	H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to -1) *
ADD, CMP/EQ	H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to -1) *
TRAPA	H'00000000 to H"000000FF	(0 to 255)

Note: * Values in the range H'FFFFFF80 to H'FFFFFFFF can be written as positive decimal values.

CAUTION!

The assembler corrects the value of displacements under certain conditions.

Condition		Type of Correction
When the operation size is a word and the displacement is not a multiple of 2	→	The lower bit of the displacement is discarded, resulting in the value being a multiple of 2.
When the operation size is a long word and the displacement is not a multiple of 4	→	The lower 2 bits of the displacement are discarded, resulting in the value being a multiple of 4.
When a displacement of the branch instruction is not a multiple of 2	→	The lower bit of the displacement is discarded, resulting in the value being a multiple of 2.

Be sure to take this correction into consideration when using operands of the mode @(disp,Rn), @(disp,GBR), and @(disp,PC).

Example: `MOV.L @(63,R0)`

The assembler corrects the 63 to be 60, and generates object code identical to that for the statement MOV.L @(60,R0), and warning number 870 occurs.

3.2 Notes on Executable Instructions

3.2.1 Notes on the Operation Size

The operation sizes that can be specified vary with the mnemonic and the addressing mode combination. Table 3-4 shows the allowable executable instruction and operation size combinations.

Table 3-4 Executable Instruction and Operation Size Combinations (part 1)

1. Data Move Instructions		Operation Sizes				
Mnemonic	**Addressing Mode**	**B**	**W**	**L**	**Default when Omitted**	
MOV	#imm,Rn	O	Δ	Δ	B	
MOV	@(disp,PC),Rn	×	O	O	L	
MOV	symbol,Rn	×	O	O	L	
MOV	Rn,Rm	×	×	O	L	
MOV	Rn,@Rm	O	O	O	L	
MOV	@Rn,Rm	O	O	O	L	
MOV	Rn,@–Rm	O	O	O	L	
MOV	@Rn+,Rm	O	O	O	L	
MOV	R0,@(disp,Rn)	O	O	O	L	
MOV	Rn,@(disp,Rm)	×	×	O	L	*1
MOV	@(disp,Rn),R0	O	O	O	L	
MOV	@(disp,Rn),Rm	×	×	O	L	*2
MOV	Rn,@(R0,Rm)	O	O	O	L	
MOV	@(R0,Rn),Rm	O	O	O	L	
MOV	R0,@(disp,GBR)	O	O	O	L	
MOV	@(disp,GBR),R0	O	O	O	L	
MOVA	@(disp,PC),R0	×	×	O	L	
MOVA	symbol,R0	×	×	O	L	
MOVT	Rn	×	×	O	L	
SWAP	Rn,Rm	O	O	×	W	
XTRCT	Rn,Rm	×	×	O	L	

Notes: 1. In this case Rn must be one of R1 to R15.
2. In this case Rm must be one of R1 to R15.

Table 3-4 Executable Instruction and Operation Size Combinations (part 1) (cont)

Symbol meanings:

Rn, Rm ..... A general register (R0 to R15)
SR ..... The status register
VBR ..... The vector base register
PR ..... The procedure register
R0 ..... The general register R0 (when only R0 can be specified)
GBR ..... The global base register
MACH, MACL ..... The accumulator register
PC ..... The program counter

imm ..... An immediate value
symbol ..... A symbol
disp ..... A displacement value

B ..... Byte
L ..... Long word (4 bytes)
W ..... Word (2 bytes)

O ..... Valid specification

× ..... Invalid specification:
The assembler regards instructions with this combination as the specification being omitted.

Δ .....The assembler regards them as extended instructions.

References:

Extended Instructions

→ Programmer's Guide, 8.2, "Extended Instructions Related to Automatic Literal Pool Generation"

Table 3-4 Executable Instruction and Operation Size Combinations (part 2)

2. Arithmetic Operation Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
ADD	Rn,Rm	×	×	O	L
ADD	#imm,Rn	×	×	O	L
ADDC	Rn,Rm	×	×	O	L
ADDV	Rn,Rm	×	×	O	L
CMP/EQ	#imm,R0	×	×	O	L
CMP/EQ	Rn,Rm	×	×	O	L
CMP/HS	Rn,Rm	×	×	O	L
CMP/GE	Rn,Rm	×	×	O	L
CMP/HI	Rn,Rm	×	×	O	L
CMP/GT	Rn,Rm	×	×	O	L
CMP/PZ	Rn	×	×	O	L
CMP/PL	Rn	×	×	O	L
CMP/STR	Rn,Rm	×	×	O	L
DIV1	Rn,Rm	×	×	O	L
DIV0S	Rn,Rm	×	×	O	L
DIV0U	(no operands)	×	×	×	—
EXTS	Rn,Rm	O	O	×	W
EXTU	Rn,Rm	O	O	×	W
MAC	@Rn+,@Rm+	×	O	×	W
MULS	Rn,Rm	×	×	O	L
MULU	Rn,Rm	×	×	O	L
NEG	Rn,Rm	×	×	O	L
NEGC	Rn,Rm	×	×	O	L
SUB	Rn,Rm	×	×	O	L
SUBC	Rn,Rm	×	×	O	L
SUBV	Rn,Rm	×	×	O	L

Table 3-4 Executable Instruction and Operation Size Combinations (part 3)

3. Logic Operation Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
AND	Rn,Rm	×	×	O	L
AND	#imm,R0	×	×	O	L
AND	#imm,@(R0,GBR)	O	×	×	B
NOT	Rn,Rm	×	×	O	L
OR	Rn,Rm	×	×	O	L
OR	#imm,R0	×	×	O	L
OR	#imm,@(R0,GBR)	O	×	×	B
TAS	@Rn	O	×	×	B
TST	Rn,Rm	×	×	O	L
TST	#imm,R0	×	×	O	L
TST	#imm,@(R0,GBR)	O	×	×	B
XOR	Rn,Rm	×	×	O	L
XOR	#imm,R0	×	×	O	L
XOR	#imm,@(R0,GBR)	O	×	×	B

Table 3-4 Executable Instruction and Operation Size Combinations (part 4)

4. Shift Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
ROTL	Rn	×	×	O	L
ROTR	Rn	×	×	O	L
ROTCL	Rn	×	×	O	L
ROTCR	Rn	×	×	O	L
SHAL	Rn	×	×	O	L
SHAR	Rn	×	×	O	L
SHLL	Rn	×	×	O	L
SHLR	Rn	×	×	O	L
SHLL2	Rn	×	×	O	L
SHLR2	Rn	×	×	O	L
SHLL8	Rn	×	×	O	L
SHLR8	Rn	×	×	O	L
SHLL16	Rn	×	×	O	L
SHLR16	Rn	×	×	O	L

Table 3-4 Executable Instruction and Operation Size Combinations (part 5)

5. Branch Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
BF	symbol	×	×	×	—
BT	symbol	×	×	×	—
BRA	symbol	×	×	×	—
BSR	symbol	×	×	×	—
JMP	@Rn	×	×	×	—
JSR	@Rn	×	×	×	—
RTS	(no operands)	×	×	×	—

Table 3-4 Executable Instruction and Operation Size Combinations (part 6)

6. System Control Instructions		Operation Sizes			
Mnemonic	Addressing Mode	B	W	L	Default when Omitted
CLRT	(no operands)	×	×	×	—
CLRMAC	(no operands)	×	×	×	—
LDC	Rn,SR	×	×	O	L
LDC	Rn,GBR	×	×	O	L
LDC	Rn,VBR	×	×	O	L
LDC	@Rn+,SR	×	×	O	L
LDC	@Rn+,GBR	×	×	O	L
LDC	@Rn+,VBR	×	×	O	L
LDS	Rn,MACH	×	×	O	L
LDS	Rn,MACL	×	×	O	L
LDS	Rn,PR	×	×	O	L
LDS	@Rn+,MACH	×	×	O	L
LDS	@Rn+,MACL	×	×	O	L
LDS	@Rn+,PR	×	×	O	L
NOP	(no operands)	×	×	×	—
RTE	(no operands)	×	×	×	—
SETT	(no operands)	×	×	×	—
SLEEP	(no operands)	×	×	×	—
STC	SR,Rn	×	×	O	L
STC	GBR,Rn	×	×	O	L
STC	VBR,Rn	×	×	O	L
STC	SR,@–Rn	×	×	O	L
STC	GBR,@–Rn	×	×	O	L
STC	VBR,@–Rn	×	×	O	L
STS	MACH,Rn	×	×	O	L
STS	MACL,Rn	×	×	O	L
STS	PR,Rn	×	×	O	L
STS	MACH,@–Rn	×	×	O	L
STS	MACL,@–Rn	×	×	O	L
STS	PR,@–Rn	×	×	O	L
TRAPA	#imm	×	×	O	L

3.2.2 Notes on Delayed Branch Instructions

The unconditional branch instructions (BRA, BSR, JMP, JSR, RTS, and RTE) are delayed branch instructions. SH microprocessors execute the delay slot instruction (the instruction directly following a branch instruction in memory) before executing the delayed branch instruction.

If an instruction inappropriate for a delay slot is specified, the assembler issues error number 150 or 151.

Table 3-5 shows the relationship between the delayed branch instruction and the delay slot instructions.

Table 3-5 Relationship between Delayed Branch Instruction and Delay Slot Instructions

		Delayed Branch					
Delay Slot		**BRA**	**BSR**	**JMP**	**JSR**	**RTS**	**RTE**
BF		×	×	×	×	×	×
BT		×	×	×	×	×	×
BRA		×	×	×	×	×	×
BSR		×	×	×	×	×	×
JMP		×	×	×	×	×	×
JSR		×	×	×	×	×	×
RTS		×	×	×	×	×	×
RTE		×	×	×	×	×	×
TRAPA		×	×	×	×	×	×
MOV	@(disp,PC),Rn	Δ	Δ	Δ	Δ	Δ	Δ
	symbol,Rn	Δ	Δ	×	×	×	×
MOVA	@(disp,PC),R0	Δ	Δ	Δ	Δ	Δ	Δ
	symbol,R0	Δ	Δ	×	×	×	×
Extended	MOV.L #imm,Rn	×	×	×	×	×	×
instructions	MOV.W #imm,Rn	×	×	×	×	×	×
	MOVA #imm,R0	×	×	×	×	×	×
Any other instruction		○	○	○	○	○	○

Symbol meanings:

○ .... Normal, i.e., the assembler generates the specified object code.

Δ .... Warning 871
Note on the value of PC: PC = <destination address for the delayed branch instruction> + 2
The assembler generates the specified object code.

× ..... Error 150 or 151
The instruction specified is inappropriate as a delay slot instruction.
The assembler generates object code with a NOP instruction (H'0009) in the object code.

CAUTION!

If the delayed branch instruction and the following instruction are coded in different sections, the assembler does not check the delay slot instruction.

References:

Extended Instructions

→ Programmer's Guide, 8.2, "Extended Instructions Related to Automatic Literal Pool Generation"

3.2.3 Notes on Address Calculations

When the operand addressing mode is PC relative with displacement, i.e., @(disp,PC), the value of PC must be taken into account in coding. The value of PC can vary depending on certain conditions.

(1) Normal Case

The value of PC is the first address in the currently executing instruction plus 4 bytes.

Examples:
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

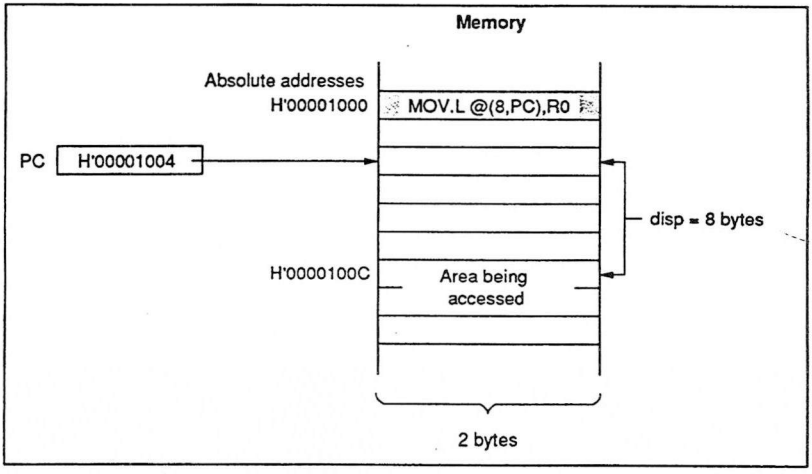


Figure 3-1 Address Calculation Example (normal case)

(2) During the Delay Slot Instruction

The value of PC is destination address for the delayed branch instruction plus 2 bytes.

Examples:
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

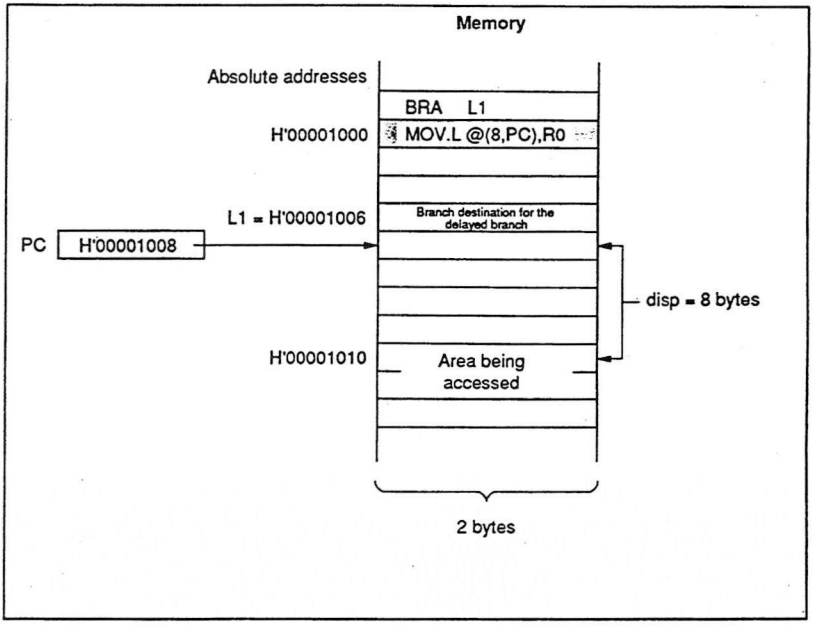


Figure 3-2 Address Calculation Example (when the value of PC differs due to a branch)

Supplement:

When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

(3) During the Execution of Either a MOV.L @(disp,PC),Rn or a MOVA @(disp,PC),R0

When the value of PC is not a multiple of 4 SH microprocessors correct the value by discarding the lower 2 bits when calculating addresses.

Examples: 1. When SH microprocessor corrects the value of PC.
(Consider the state when a MOV instruction is being executed at absolute address H'00001002.)

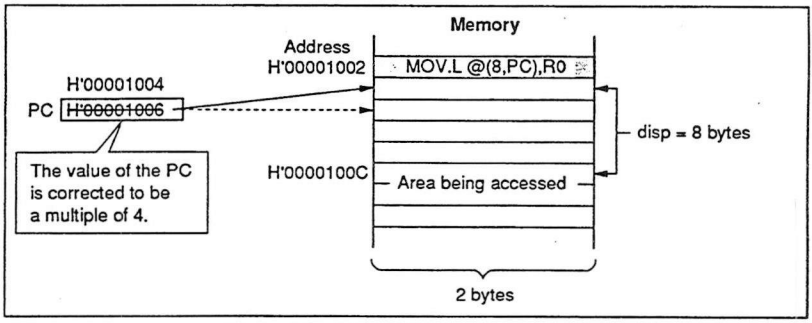


Figure 3-3 Address Calculation Example (when SH microprocessor corrects the value of PC)

2. When SH microprocessor does not correct the value of PC.
(Consider the state when a MOV instruction is being executed at absolute address H'00001000.)

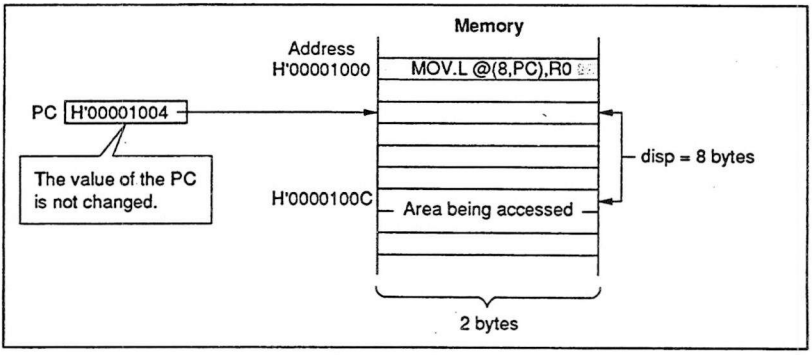


Figure 3-4 Address Calculation Example (when SH microprocessor does not correct the value of PC)

Supplement:

When the operand is the PC relative specified with the symbol, the assembler derives the displacement taking account of the value of PC when generating the object code.

Section 4 Assembler Directives

4.1 Overview of the Assembler Directives

The assembler directives are instructions that the assembler interprets and executes. Table 4-1 lists the assembler directives provided by this assembler.

Table 4-1 Assembler Directives

Type	Mnemonic	Function
Section and the location counter	.SECTION	Declares a section.
	.ORG	Sets the value of the location counter.
	.ALIGN	Corrects the value of the location counter.
Symbols	.EQU	Sets a symbol value (reset not allowed).
	.ASSIGN	Sets a symbol value (reset allowed).
	.REG	Defines the alias of a register name.
Data and data area reservation	.DATA	Reserves integer data.
	.DATAB	Reserves integer data blocks.
	.SDATA	Reserves character string data.
	.SDATAB	Reserves character string data blocks.
	.SDATAC	Reserves character string data (with length).
	.SDATAZ	Reserves character string data (with zero terminator).
	.RES	Reserves data area.
	.SRES	Reserves character string data area.
	.SRESC	Reserves character string data area (with length).
	.SRESZ	Reserves character string data area (with zero terminator).
Export and import symbol	.EXPORT	Declares export symbols.
	.IMPORT	Declares import symbols.
	.GLOBAL	Declares export and import symbols.
Object modules	.OUTPUT	Controls object module output.
	.DEBUG	Controls the output of symbolic debug information.

Table 4-1 Assembler Directives (cont)

Type	Mnemonic	Function
Assemble listing	.PRINT	Controls assemble listing output.
	.LIST	Controls the output of the source program listing.
	.FORM	Sets the number of lines and columns in the assemble listing.
	.HEADING	Sets the header for the source program listing.
	.PAGE	Inserts a new page in the source program listing.
	.SPACE	Outputs blank lines to the source program listing.
Other directives	.PROGRAM	Sets the name of the object module.
	.RADIX	Sets the radix in which integer constants with no radix specifier are interpreted.
	.END	Declares the end of the source program.

4.2 Assembler Directive Reference

4.2.1 Section and Location Counter Assembler Directives

This assembler provides the following assembler directives concerned with sections and the location counter.

.SECTION

Declares a section.

.ORG

Sets the value of the location counter.

.ALIGN

Adjusts the value of the location counter to a multiple of the boundary alignment value.

.SECTION

Section Declaration

Syntax

```
.SECTIONΔ<section name> [,<section attribute> [,{LOCATE=
<start address>|ALIGN=<boundary alignment value>}]]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .SECTION mnemonic in the operation field.

3. Operands

   a. First operand: the section name

      The rules for section names are the same as the rules for symbols.

      References: Naming sections
      → Programmer's Guide, 1.3.2, "Coding of Symbols"

   b. Second operand: the section attribute

Attribute	Section Type
CODE	Code section
DATA	Data section
STACK	Stack section
COMMON	Common section
DUMMY	Dummy section

The shaded section indicates the default value when the specifier is omitted.

When the specification is omitted, the section will be a code section.

c. Third operand: start address or boundary alignment value

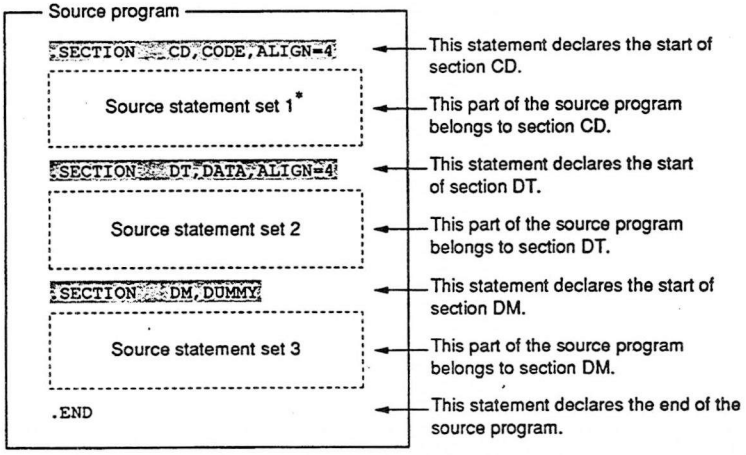
Specification	Section Type
LOCATE = <start address>	Absolute address section
ALIGN = <boundary alignment value>	Relative address section

The specification determines whether the section type will be an absolute address section or a relative address section. When the specification is omitted, the section will be a relative address section with boundary alignment value of 4.

Description

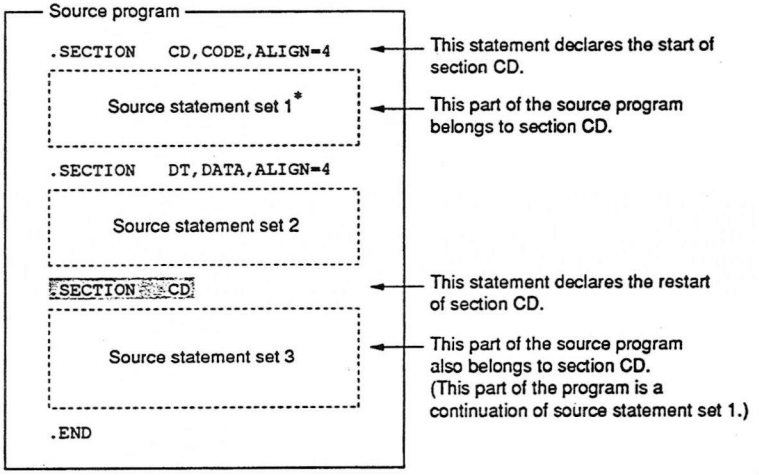
1. .SECTION is the section declaration assembler directive.

A section is a part of a program, and the linkage editor regards it as a unit of processing. The following describes section declaration using the simple examples shown below.



Note: * This example assumes that the .SECTION assembler directive does not appear in any of the source statement sets 1 to 3.

2. It is possible to redeclare (and thus restart, i.e., re-enter) a section that was previously declared in the same file. The following is a simple example of section restart.



Note: * This example assumes that the .SECTION assembler directive does not appear in any of the source statement sets 1 to 3.

CAUTION!

When using the .SECTION assembler directive to restart a section, the second and third operands must be omitted. (The original specifications when first declaring the section remain valid.)

3. Use LOCATE = <start address> as the third operand when starting an absolute address section. The start address is the absolute address of the start of that section.

   The start address must be specified as follows:

   - The specification must be an absolute value, and,
   - Forward reference symbols must not appear in the specification.

The values allowed for the start address are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

4. Use ALIGN = <boundary alignment value> to start a relative address section. The linkage editor will adjust the start address of the section to be a multiple of the boundary alignment value.

   The boundary alignment value must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the boundary alignment value are powers of 2, e.g. 2^0, 2^1, 2^2, ..., 2^{31}. For code sections, the values must be 4 or larger powers of 2, e.g. 2^2, 2^3, 2^4, ..., 2^{31}.

5. The assembler provides a default section for the following cases.

   - The use of executable instructions when no section has been declared.
   - The use of data reservation assembler directives when no section has been declared.
   - The use of the .ALIGN assembler directive when no section has been declared.
   - The use of the .ORG assembler directive when no section has been declared.
   - Reference to the location counter when no section has been declared.
   - The use of statements consisting of only the label field when no section has been declared.

   The default section is the following section.

   - Section name: P
   - Section type: Code section
     Relative address section (with a boundary alignment value of 4)

.SECTION

Coding Example

```
                .ALIGN      4                   ; This section of the program belongs to the default section P.
                .DATA.L     H'11111111          ; The default section P is a code section, and is a relative
                  ~                             ; address section with a boundary alignment value of 4.

.SECTION CD,CODE,ALIGN=4

                MOV         R0,R1               ; This section of the program belongs to the section CD.
                MOV         R0,R2               ; The section CD is a code section, and is a relative address
                  ~                             ; section with a boundary alignment value of 4.

.SECTION DT,DATA,LOCATE=H'00001000

X1:             .DATA.L     H'22222222          ; This section of the program belongs to the section DT.
                .DATA.L     H'33333333          ; The section DT is a data section, and is an absolute address
                  ~                             ; section with a start address of H'00001000.

                .END
```

Note: This example assumes the .SECTION assembler directive does not appear in the parts indicated by "~".

Location-Counter-Value Setting

Syntax

```
.ORGΔ<location-counter-value>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .ORG mnemonic in the operation field.

3. Operands

   Enter the new value for the location counter in the operand field.

Description

1. .ORG is an assembler directive that sets the value of the location counter. The .ORG assembler directive is used to place executable instructions or data at a specific address.

2. The location-counter-value must be specified as follows:

   - The specification must be an absolute value or an address within the section, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the location-counter-value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

   When the location-counter-value is specified with an absolute value, the following condition must hold:

   <location-counter-value> ≥ <section start address> (when compared as unsigned values)

3. The assembler handles the value of the location counter as follows.

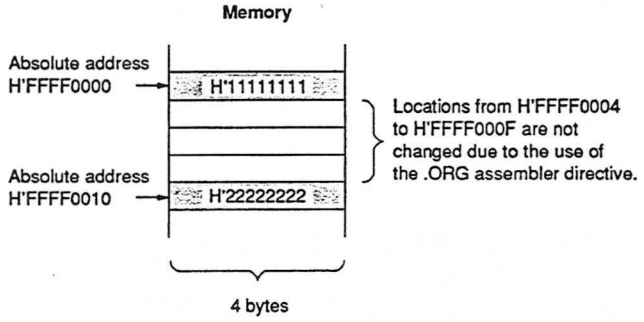
   - The value is regarded as an absolute address value within an absolute address section.
   - The value is regarded as a relative address value within a relative address section.

Coding Example

```
.SECTION   DT,DATA,LOCATE=H'FFFF0000
.DATA.L    H'11111111

.ORG       H'FFFF0010      ; This statement sets the value of the location
                           counter.
.DATA.L    H'22222222      ; The integer data H'22222222 is stored at
                           ; absolute address H'FFFF0010.
```

Explanatory Figure for the Coding Example



Location-Counter-Value Correction

Syntax

```
.ALIGNΔ<boundary alignment value>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .ALIGN mnemonic in the operation field.

3. Operands

   Enter the boundary alignment value in the operand field.

Description

1. .ALIGN is an assembler directive that corrects the location-counter-value to be a multiple of the boundary alignment value. Executable instructions and data can be allocated on specific boundary values (address multiples) by using the .ALIGN assembler directive.

2. The boundary alignment value must be specified as follows:

   - The specification must be an absolute value, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the boundary alignment value are powers of 2, e.g. 2^0, 2^1, 2^2, ..., 2^{31}.

   The boundary alignment value specified by .ALIGN directive must be less than or equal to the boundary alignment value specified by .SECTION directive.

3. When .ALIGN is used in a code section, data section, or dummy section, the assembler inserts NOP instructions in the object code* to adjust the value of the location counter. Odd byte size areas are filled with H'09.

.ALIGN

When .ALIGN is used in a dummy or stack section, the assembler only adjusts the value of the location counter, and does not fill in any object code in memory.

Note: * This object code is not displayed in the assemble listing.

Coding Example

```
        ~

.DATA.B     H'11
.DATA.B     H'22
.DATA.B     H'33

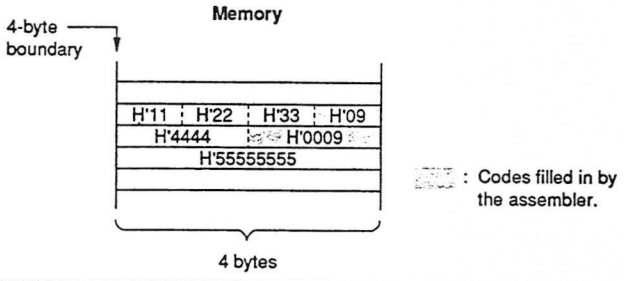
.ALIGN      2               ; This statement adjusts the value of the location
.DATA.W     H'4444          ; counter to be a multiple of 2.

.ALIGN      4               ; This statement adjusts the value of the location
.DATA.L     H'55555555      ; counter to be a multiple of 4.

        ~
```

Explanatory Figure for the Coding Example

This example assumes that the byte sized integer data H'11 is originally located at the 4-byte boundary address. The assembler will insert the filler data as shown in the figure below.



4.2.2 Symbol Handling Assembler Directives

This assembler provides the following assembler directives concerned with symbols.

.EQU

Sets a symbol value.

.ASSIGN

Sets and resets a symbol value.

.REG

Defines the alias of a register name.

.EQU

Symbol Value Setting (resetting not allowed)

Syntax

```
<symbol>[:]Δ.EQUΔ<symbol value>
```

Statement Elements

1. Label

   Enter the symbol that is to be set a value in the label field.

2. Operation

   Enter the .EQU mnemonic in the operation field.

3. Operands

   Enter the value to be set to the symbol in the operand field.

Description

1. .EQU is an assembler directive that sets a value to a symbol.

   Symbols defined with the .EQU directive cannot be redefined.

2. The symbol value must be specified as follows:

   - The specification must be an absolute value or an address value, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the symbol value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

Coding Example

```
          ~

X1:     .EQU      10          ; The value 10 is set to X1.
X2:     .EQU      20          ; The value 20 is set to X2.

        CMP/EQ    #X1,R0      ; This is the same as CMP/EQ #10,R0.
        BT        LABEL1
        CMP/EQ    #X2,R0      ; This is the same as CMP/EQ #20,R0.
        BT        LABEL2

          ~
```

.ASSIGN

Symbol Value Setting (resetting allowed)

Syntax

```
<symbol>[:]Δ.ASSIGNΔ<symbol value>
```

Statement Elements

1. Label

   Enter the symbol that is to be set a value in the label field.

2. Operation

   Enter the .ASSIGN mnemonic in the operation field.

3. Operands

   Enter the value to be set to the symbol in the operand field.

Description

1. .ASSIGN is an assembler directive that sets a value to a symbol.

   Symbols defined with the .ASSIGN directive can be redefined with the .ASSIGN directive.

2. The symbol value must be specified as follows:

   - The specification must be an absolute value or an address value, and,
   - Forward reference symbols must not appear in the specification.

   The values allowed for the symbol value are from H'00000000 to H'FFFFFFFF. (From –2,147,483,648 to 4,294,967,295 in decimal.)

3. Definitions with the .ASSIGN directive are valid from the point of the definition forward in the program.

4. Symbols defined with .ASSIGN have the following limitations:

   - They cannot be used as export or import symbols.
   - They cannot be referenced from the simulator/debugger.

Coding Example

```
            ~

X1:         .ASSIGN   1
X2:         .ASSIGN   2
            CMP/EQ    #X1,R0      ; This is the same as CMP/EQ #1,R0.
            BT        LABEL1
            CMP/EQ    #X2,R0      ; This is the same as CMP/EQ #2,R0.
            BT        LABEL2

            ~

X1:         .ASSIGN   3
X2:         .ASSIGN   4
            CMP/EQ    #X1,R0      ; This is the same as CMP/EQ #3,R0.
            BT        LABEL3
            CMP/EQ    #X2,R0      ; This is the same as CMP/EQ #4,R0.
            BF        LABEL4

            ~
```

.REG

Alias of a Register Name Definition

Syntax

```
<symbol>[:]Δ.REGΔ(<register name>)
```

Statement Elements

1. Label

   Enter the symbol to be defined as the alias of a register name in the label field.

2. Operation

   Enter the .REG mnemonic in the operation field.

3. Operands

   Enter the register name for which the alias of a register name is being defined in the operand field.

Description

1. .REG is the assembler directive that defines the alias of a register name.

   - The alias of a register name defined with .REG can be used in exactly the same manner as the original register name.

     The alias of a register name defined with .REG cannot be redefined.

2. The alias of a register name can only be defined for the general registers (R0 to R15, and SP).

3. Definitions with the .REG directive are valid from the point of the definition forward in the program.

4. Symbols defined with .REG have the following limitations:

   - They cannot be used as export or import symbols.
   - They cannot be referenced from the simulator/debugger.

Coding Example

```
            ~

MIN:    .REG    (R10)
MAX:    .REG    (R11)

        MOV     #0,MIN      ; This is the same as MOV #0,R10.
        MOV     #99,MAX     ; This is the same as MOV #99,R11.

        CMP/HS  MIN,R1
        BF      LABEL
        CMP/HS  R1,MAX
        BF      LABEL

            ~
```

4.2.3 Data and Data Area Reservation Assembler Directives

This assembler provides the following assembler directives that are concerned with data and data area reservation.

.DATA	Reserves integer data.
.DATAB	Reserves integer data blocks.
.SDATA	Reserves character string data.
.SDATAB	Reserves character string data blocks.
.SDATAC	Reserves character string data (with length).
.SDATAZ	Reserves character string data (with zero terminator).
.RES	Reserves data area.
.SRES	Reserves character string data area.
.SRESC	Reserves character string data area (with length).
.SRESZ	Reserves character string data area (with zero terminator).

Integer Data Reservation

Syntax

```
[<symbol>[:]]Δ.DATA[.<operation size>]Δ<integer data>[,<integer data>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .DATA mnemonic in the operation field.

   b. Operation size

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

   The shaded section indicates the default value when the specifier is omitted.

   The specifier determines the size of the reserved data.
   The long word size is used when the specifier is omitted.

3. Operands

   Enter the values to be reserved as data in the operand field.

Description

1. .DATA is the assembler directive that reserves integer data in memory.
2. Arbitrary values, including relative values and forward reference symbols, can be used to specify the integer data.

3. The range of values that can be specified as integer data varies with the operation size.

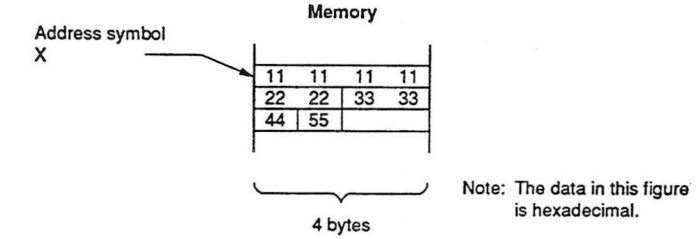
Operation Size	Integer Data Range*	
B	H'00000000 to H'000000FF	(0 to 255)
	H'FFFFFF80 to H'FFFFFFFF	(–128 to -1)
W	H'00000000 to H'0000FFFF	(0 to 65,535)
	H'FFFF8000 to H'FFFFFFFF	(–32,768 to -1)
L	H'00000000 to H'7FFFFFFF	(0 to 4,294,967,295)
	H'80000000 to H'FFFFFFFF	(–2,147,483,648 to -1)

Note: * Numbers in parentheses are decimal.

Coding Example

```
            ~
        .ALIGN    4              ; (This statement adjusts the value of the
                                   location counter.)
X:      .DATA.L   H'11111111     ;
        .DATA.W   H'2222,H'3333  ; These statements reserve integer data.
        .DATA.B   H'44,H'55      ;
            ~
```

Explanatory Figure for the Coding Example



Integer Data Block Reservation

Syntax

```
[<symbol>[:]]Δ.DATAB[.<operation size>]Δ<block count>,<integer data>
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .DATAB mnemonic in the operation field.

   b. Operation size

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

The shaded section indicates the default value when the specifier is omitted.

   The specifier determines the size of the reserved data.
   The long word size is used when the specifier is omitted.

3. Operands

   a. First operand: block count
   Enter the number of times the data value is repeated as the first operand.

   b. Second operand: integer data
   Enter the value to be reserved as the second operand.

.DATAB

Description

1. .DATAB is the assembler directive that reserves the specified number of integer data consecutively in memory.

2. The block count must be specified as follows:

   - The specification must be an absolute value, and,
   - Forward reference symbols must not appear in the specification.

   Arbitrary values, including relative values and forward reference symbols, can be used to specify the integer data.

3. The range of values that can be specified as the block size and as the integer data varies with the operation size.

Operation Size	Block Size Range*	
B	H'00000001 to H'FFFFFFFF	(1 to 4,294,967,295)
W	H'00000001 to H'7FFFFFFF	(1 to 2,147,483,647)
L	H'00000001 to H'3FFFFFFF	(1 to 1,073,741,823)

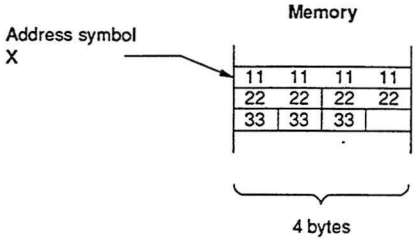
Operation Size	Integer Data Range*	
B	H'00000000 to H'000000FF H'FFFFFF80 to H'FFFFFFFF	(0 to 255) (–128 to -1)
W	H'00000000 to H'0000FFFF H'FFFF8000 to H'FFFFFFFF	(0 to 65,535) (–32,768 to -1)
L	H'00000000 to H'7FFFFFFF H'80000000 to H'FFFFFFFF	(0 to 4,294,967,295) (–2,147,483,648 to -1)

Note: * Numbers in parentheses are decimal.

Coding Example

```
                ~
        .ALIGN      4               ; (This statement adjusts the value of the
                                    ; location counter.)
X:      .DATAB.L    1,H'11111111    ;
        .DATAB.W    2,H'2222        ; This statement reserves two blocks of integer
        .DATAB.B    3,H'33          ; data.
                ~
```

Explanatory Figure for the Coding Example



Note: The data in this figure is hexadecimal.

.SDATA

Character String Data Reservation

Syntax

```
[<symbol>[:]]Δ.SDATAΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATA mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATA is the assembler directive that reserves character string data in memory.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

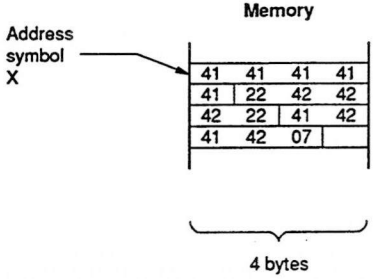
   The ASCII code for a control character must be specified as follows.

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

Coding Example

```
            ~
            .ALIGN      4               ; (This statement adjusts the value of
                                        ; the location counter.)
X:          .SDATA      "AAAAA"         ; This statement reserves character string data.
            .SDATA      """BBB"""       ; The character string in this example includes
                                        ; double quotation marks.
            .SDATA      "ABAB"<H'07>    ; The character string in this example has
                                        ; a control character appended.
            ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'4
The ASCII code for "B" is: H'4
The ASCII code for """ is: H'2

.SDATAB

Character String Data Blocks Reservation

Syntax

```
[<symbol>[:]]Δ.SDATABΔ<block count>,"<character string>"
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAB mnemonic in the operation field.

3. Operands

   a. First operand: <block count>
      Enter the number of character strings as the first operand.

   b. Second operand: <character string>
      Enter the character string to be reserved as the second operand.

Description

1. .SDATAB is the assembler directive that reserves the specified number of character strings consecutively in memory.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The <block count> must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   A value of 1 or larger must be specified as the block count.

   The maximum value of the block count depends on the length of the character string data.

(The length of the character string data multiplied by the block count must be less than or equal to H'FFFFFFFF (4,294,967,295) bytes.)

3. A control character can be appended to a character string.

   The syntax for this notation is as follows.

```
"<character string>"<<ASCII code for a control character>>
```

   The ASCII code for a control character must be specified as follows.

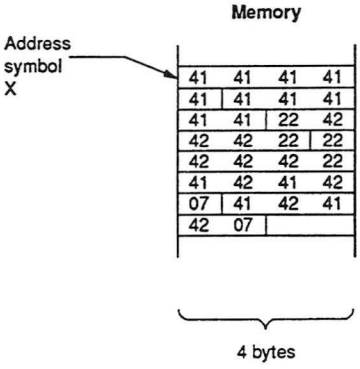
   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

.SDATAB

Coding Example

```
            ~
            .ALIGN      4               ; (This statement adjusts the value of the
                                        ; location counter.)
X:          .SDATAB     2,"AAAAA"       ; This statement reserves two character string
                                        ; data blocks.
            .SDATAB     2,"""BBB"""     ; The character string in this example includes
                                        ; double quotation marks.
            .SDATAB     2,"ABAB"<H'07>  ; The character string in this example has
                                        ; a control character appended.
            ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for """ is: H'22.

Character String Data Reservation (with length)

Syntax

```
[<symbol>[:]]Δ.SDATACΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAC mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATAC is the assembler directive that reserves character string data (with length) in memory.

   A character string with length is a character string with an inserted leading byte that indicates the length of the string.

   The length indicates the size of the character string (not including the length) in bytes.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

   The ASCII code for a control character must be specified as follows.

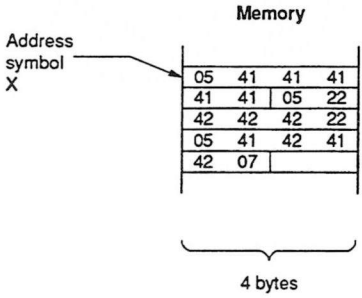
.SDATAC

- The specification must be an absolute value, and,
- Forward reference symbols must not appear in the specification.

Coding Example

```
            ~
            .ALIGN      4               ; (This statement adjusts the value of the
                                        ; location counter.)
X:          .SDATAC     "AAAAA"         ; This statement reserves character string data
                                        ; (with length).
            .SDATAC     """BBB"""       ; The character string in this example includes
                                        ; double quotation marks.
            .SDATAC     "ABAB"<H'07>    ; The character string in this example has
                                        ; a control character appended.
            ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for "" is: H'22.

Character String Data Reservation (with zero terminator)

Syntax

```
[<symbol>[:]]Δ.SDATAZΔ"<character string>"[,"<character string>"...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SDATAZ mnemonic in the operation field.

3. Operands

   Enter the character string(s) to be reserved in the operand field.

Description

1. .SDATAZ is the assembler directive that reserves character string data (with zero terminator) in memory.

   A character string with zero terminator is a character string with an appended trailing byte (with the value H'00) that indicates the end of the string.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. A control character can be appended to a character string.

   The syntax for this notation is as follows.

   ```
   "<character string>"<<ASCII code for a control character>>
   ```

   The ASCII code for a control character must be specified as follows.

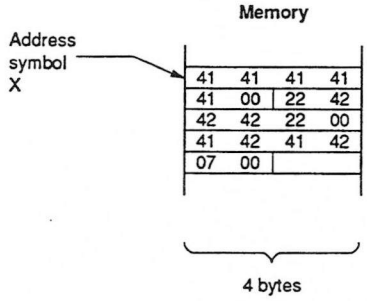
.SDATAZ

- The specification must be an absolute value, and,
- Forward reference symbols must not appear in the specification.

Coding Example

```
            ~
            .ALIGN      4               ; (This statement adjusts the value of the
                                        ; location counter.)
X:          .SDATAZ     "AAAAA"         ; This statement reserves character string
                                        ; data (with zero terminator).
            .SDATAZ     """BBB"""       ; The character string in this example includes
                                        ; double quotation marks.
            .SDATAZ     "ABAB"<H'07>    ; The character string in this example has
                                        ; a control character appended.
            ~
```

Explanatory Figure for the Coding Example



Notes: 1. The data in this figure is hexadecimal.

2. The ASCII code for "A" is: H'41.
The ASCII code for "B" is: H'42.
The ASCII code for """ is: H'22.

Data Area Reservation

Syntax

```
[<symbol>[:]]Δ.RES[.<operation size>]Δ<area count>
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   a. Mnemonic
   Enter .RES mnemonic in the operation field.

   b. Operation size

Specifier	Data Size
B	Byte
W	Word (2 bytes)
L	Long word (4 bytes)

   The shaded section indicates the default value when the specifier is omitted.

   The specifier determines the size of one area.
   The long word size is used when the specifier is omitted.

3. Operands

   Enter the number of areas to be reserved in the operand field.

.RES

Description

1. .RES is the assembler directive that reserves data areas in memory.
2. The area count must be specified as follows:
   - The specification must be an absolute value, and,
   - Forward reference symbols must not appear in the specification.
3. The range of values that can be specified as the area count varies with the operation size.

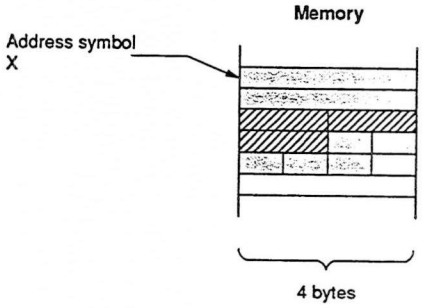
Operation Size	Area Count Range*	
B	H'00000001 to H'FFFFFFFF	(1 to 4,294,967,295)
W	H'00000001 to H'7FFFFFFF	(1 to 2,147,483,647)
L	H'00000001 to H'3FFFFFFF	(1 to 1,073,741,823)

Note: * Numbers in parentheses are decimal.

Coding Example

```
        ~
        .ALIGN      4       ; (This statement adjusts the value of the location
                            ; counter.)
X:      .RES.L      2       ; This statement reserves 2 long word size areas.
        .RES.W      3       ; This statement reserves 3 word size areas.
        .RES.B      5       ; This statement reserves 5 byte size areas.
        ~
```

Explanatory Figure for the Coding Example



.SRES

Character String Data Area Reservation

Syntax

```
[<symbol>[:]]Δ.SRESΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRES mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas to be reserved in the operand field.

Description

1. .SRES is the assembler directive that reserves character string data areas.

2. The character string area size must be specified as follows:

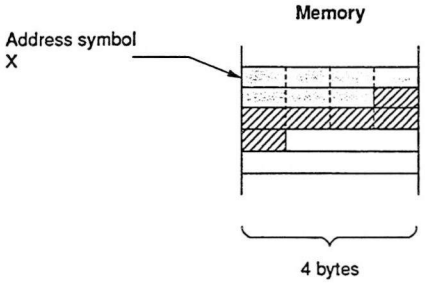
   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   The values that are allowed for the character string area size are from H'00000001 to H'FFFFFFFF (from 1 to 4,294,967,295 in decimal).

Coding Example

```
                ~
        .ALIGN      4       ; (This statement adjusts the value of the location
                            ; counter.)
X:      .SRES       7       ; This statement reserves a 7-byte area.
        .SRES       6       ; This statement reserves a 6-byte area.
                ~
```

Explanatory Figure for the Coding Example



.SRESC

Character String Data Area Reservation (with length)

Syntax

```
[<symbol>[:]]Δ.SRESCΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRESC mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas (not including the length) to be reserved in the operand field.

Description

1. .SRESC is the assembler directive that reserves character string data areas (with length) in memory.

   A character string with length is a character string with an inserted leading byte that indicates the length of the string.

   The length indicates the size of the character string (not including the length) in bytes.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The character string area size must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

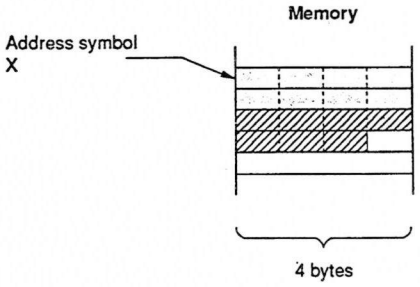
   The values that are allowed for the character string area size are from H'00000000 to H'000000FF (in decimal, from 0 to 255).

3. The size of the area reserved in memory is the size of the character string area itself plus 1 byte for the count.

Coding Example

```
            ~
            .ALIGN      4       ; (This statement adjusts the value of the location
                                ; counter.)
X:          .SRESC      7       ; This statement reserves 7 bytes plus 1 byte for
                                ; the count.
            .SRESC      6       ; This statement reserves 6 bytes plus 1 byte for
                                ; the count.
            ~
```

Explanatory Figure for the Coding Example



.SRESZ

Character String Data Area Reservation (with zero terminator)

Syntax

```
[<symbol>[:]]Δ.SRESZΔ<character string area size>[,<character string area size>...]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the .SRESZ mnemonic in the operation field.

3. Operands

   Enter the sizes of the areas (not including the terminating zero) to be reserved in the operand field.

Description

1. .SRESZ is the assembler directive that allocates character string data areas (with zero termination).

   A character string with length is a character string with an appended trailing byte (with the value H'00) that indicates the end of the string.

   References: Character strings → Programmer's Guide, 1.7, "Character Strings"

2. The character string area size must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

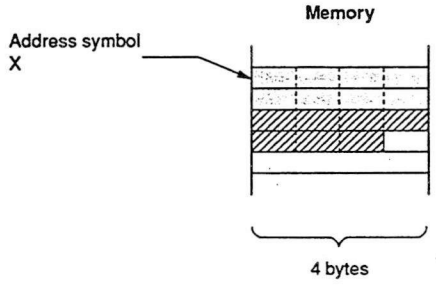
   The values that are allowed for the character string area size are from H'00000000 to H'000000FF (in decimal, from 0 to 255).

3. The size of the area reserved in memory is the size of the character string area itself plus 1 byte for the terminating zero.

Coding Example

```
            ~
            .ALIGN      4       ; (This statement adjusts the value of the location counter.)
X:          .SRESZ      7       ; This statement reserves 7 bytes plus 1 byte for
                                ; the terminating byte.
            .SRESZ      6       ; This statement reserves 6 bytes plus 1 byte for
                                ; the terminating byte.
            ~
```

Explanatory Figure for the Coding Example



4.2.4 Export and Import Assembler Directives

This assembler provides the following assembler directives concerned with export and import.

```
.EXPORT
```

Declares export symbols.

This declaration allows symbols defined in the current file to be referenced in other files.

```
.IMPORT
```

Declares import symbols.

This declaration allows symbols defined in other files to be referenced in the current file.

```
.GLOBAL
```

Declares export and import symbols.

This declaration allows symbols defined in the current file to be referenced in other files, and allows symbols defined in other files to be referenced in the current file.

Export Symbols Declaration

Syntax

```
.EXPORTΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXPORT mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as export symbols in the operand field.

Description

1. .EXPORT is the assembler directive that declares export symbols.

   An export symbol declaration is required to reference symbols defined in the current file from other files.

2. The following can be declared to be export symbols.

   - Constant symbols (other than those defined with the .ASSIGN assembler directive)
   - Absolute address symbols (other than address symbols in a dummy section)
   - Relative address symbols

3. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Import symbols are declared in the file in which they are referenced using either the .IMPORT or the .GLOBAL assembler directive.

.EXPORT

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
        .EXPORT   MAX              ; This statement declares X to be an export
                                   ; symbol.
          ~
X:      .EQU      H'10000000       ; This statement defines X.
          ~
```

File B:

```
        .IMPORT   X                ; This statement declares X to be an import
                                   ; symbol.
          ~
        .ALIGN    4
        .DATA.L   X                ; This statement references X.
          ~
```

Import Symbols Declaration

Syntax

```
.IMPORTΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .IMPORT mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as import symbols in the operand field.

Description

1. .IMPORT is the assembler directive that declares import symbols.

   An import symbol declaration is required to reference symbols defined in another file.

2. Symbols defined in the current file cannot be declared to be import symbols.

3. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Export symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL assembler directive.

.IMPORT

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
        .EXPORT   X                ; This statement declares X to be an export
                                   ; symbol.
          ~

X:      .EQU      H'10000000       ; This statement defines X.

          ~
```

File B:

```
        .IMPORT   X                ; This statement declares X to be an import
                                   ; symbol.
          ~

        .ALIGN    4
        .DATA.L   X                ; This statement references X.

          ~
```

Export and Import Symbols Declaration

Syntax

```
.GLOBALΔ<symbol>[,<symbol>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .GLOBAL mnemonic in the operation field.

3. Operands

   Enter the symbols to be declared as export symbols or as import symbols in the operand field.

Description

1. .GLOBAL is the assembler directive that declares symbols to be either export symbols or import symbols.

   An export symbol declaration is required to reference symbols defined in the current file from other files. An import symbol declaration is required to reference symbols defined in another file.

2. A symbol defined within the current file is declared to be an export symbol by a .GLOBAL declaration.

   A symbol that is not defined within the current file is declared to be an import symbol by a .GLOBAL declaration.

3. The following can be declared to be export symbols.

   - Constant symbols (other than those defined with the .ASSIGN assembler directive)

- Absolute address symbols (other than address symbols in a dummy section)
- Relative address symbols

4. To reference a symbol as an import symbol, it is necessary to declare it to be an export symbol, and also to declare it to be an import symbol.

   Export symbols are declared in the file in which they are defined using either the .EXPORT or the .GLOBAL assembler directive.

   Import symbols are declared in the file in which they are referenced using either the .IMPORT or the .GLOBAL assembler directive.

Coding Example

(In this example, a symbol defined in file A is referenced from file B.)

File A:

```
        .GLOBAL  X              ; This statement declares X to be an export
                                ; symbol.
          ~

X:      .EQU     H'10000000     ; This statement defines X.

          ~
```

File B:

```
        .GLOBAL  X              ; This statement declares X to be an import
                                ; symbol.
          ~

        .ALIGN   4
        .DATA.L  X              ; This statement references X.

          ~
```

4.2.5 Object Module Assembler Directives

This assembler provides the following assembler directives concerned with object modules.

.OUTPUT

Controls object module and debug information output.

.DEBUG

Controls the output of symbolic debug information.

```
.OUTPUT
```

Object Module Output Control

Syntax

```
.OUTPUTΔ<output specifier>[,<output specifier>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .OUTPUT mnemonic in the operation field.

3. Operands: <output specifier>

Output Specifier	Output Control
OBJ	An object module is output.
NOOBJ	No object module is output.
DBG	Debug information is output in the object module.
NODBG	No debug information is output in the object module.

The shaded section indicates the default value when the specifier is omitted.

The output specifiers control object module and debug information output.

Description

1. .OUTPUT is the assembler directive that controls object module and debug information output.

2. If the .OUTPUT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

Example:

```
   ~
.OUTPUT  OBJ
.OUTPUT  NODBG   ← OK
   ~
```

```
   ~
.OUTPUT  OBJ
.OUTPUT  NOOBJ   ← Error
   ~
```

3. Specifications concerning debug information output are only valid when an object module is output.

4. The assembler gives priority to command line option specifications concerning object module and debug information output.

References: Object module output
→ User's Guide, 2.2.1, "Object Module Command Line Options" OBJECT NOOBJECT

Debug information output
→ User's Guide, 2.2.1, "Object Module Command Line Options" DEBUG NODEBUG

Coding Example

Note: This example and its description assume that no command line options concerning object module or debug information output were specified.

```
        .OUTPUT     OBJ             ; An object module is output.
                                    ; No debug information is output.
          ~

-------------------------------------------------------------------------

        .OUTPUT     OBJ,DBG         ; Both an object module and debug information
                                    ; is output.
          ~

-------------------------------------------------------------------------

        .OUTPUT     OBJ,NODBG       ; An object module is output.
                                    ; No debug information is output.
          ~
```

.OUTPUT

Supplement:

Debug information is required when debugging a program using the simulator/debugger, and is part of the object module.

Debug information includes information about source statements and information about symbols.

Symbolic Debug Information Output Control

Syntax

```
.DEBUGΔ<output specifier>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .DEBUG mnemonic in the operation field.

3. Operands: output specifier

Output Specifier	Output Control
ON	Symbolic debug information is output starting with the next source statement.
OFF	Symbolic debug information is not output starting with the next source statement.

The shaded section indicates the default value when the specifier is omitted.

The output specifier controls symbolic debug information output.

Description

1. .DEBUG is the assembler directive that controls the output of symbolic debug information.

   This directive allows assembly time to be reduced by restricting the output of symbolic debug information to only those symbols required in debugging.

2. The specification of the .DEBUG directive is only valid when both an object module and debug information are output.

.DEBUG

References: Object module output
→ Programmer's Guide, 4.2.5, "Object Module Assembler Directives", .OUTPUT
→ User's Guide, 2.2.1 "Object Module Command Line Options" OBJECT NOOBJECT

Debug information output
→ Programmer's Guide 4.2.5, "Object Module Assembler Directives", .OUTPUT
→ User's Guide, 2.2.1, "Object Module Command Line Options" DEBUG NODEBUG

Coding Example

```
        ~
.DEBUG  OFF     ; Starting with the next statement, the assembler
                ; does not output symbolic debug information.
        ~
.DEBUG  ON      ; Starting with the next statement, the assembler
                ; outputs symbolic debug information.
        ~
```

Supplement:

The term "symbolic debug information" refers to the parts of debug information concerned with symbols.

4.2.6 Assemble Listing Assembler Directives

This assembler provides the following assembler directives for controlling the assemble listing.

.PRINT	Controls assemble listing output.
.LIST	Controls the output of the source program listing.
.FORM	Sets the number of lines and columns in the assemble listing.
.HEADING	Sets the header for the source program listing.
.PAGE	Inserts a new page in the source program listing.
.SPACE	Outputs blank lines to the source program listing.

Supplement:

The assemble listing is a listing to which the results of the assembly are output, and includes a source program listing, a cross-reference listing, and a section information listing.

References: For a detailed description of the assemble listing, see appendix C, "Assemble Listing Output Example".

.PRINT

Assemble Listing Output Control

Syntax

```
.PRINTΔ<output specifier>[,<output specifier>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PRINT mnemonic in the operation field.

3. Operands: output specifier

Output Specifier	Assembler Action
LIST	An assemble listing is output.
NOLIST	No assemble listing is output.
SRC	A source program listing is output in the assemble listing.
NOSRC	No source program listing is output in the assemble listing.
CREF	A cross-reference listing is output in the assemble listing.
NOCREF	No cross-reference listing is output in the assemble listing.
SCT	A section information listing is output in the assemble listing.
NOSCT	No section information listing is output in the assemble listing.

The shaded sections indicate the default settings when the specifier is omitted.

The output specifier controls assemble listing output.

Description

1. .PRINT is the assembler directive that controls assemble listing output.
2. If the .PRINT directive is used two or more times in a program with inconsistent output specifiers, an error occurs.

   Example:

```
    ~
.PRINT  LIST
.PRINT  NOSRC      ← OK
    ~
```

```
    ~
.PRINT  LIST
.PRINT  NOLIST     ← Error
    ~
```

3. The output specifiers concerned with the source program listing, the cross-reference listing, and the section information listing are only valid when an assemble listing is output.
4. The assembler gives priority to command line option specifications concerning assemble listing output.

   References: Assemble listing output
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options"
   LIST NOLIST
   SOURCE NOSOURCE
   CROSS_REFERENCE NOCROSS_REFERENCE
   SECTION NOSECTION

Coding Example

Note: This example and its description assume that no command line options concerning assemble listing output are specified.

```
        .PRINT    LIST                  ; All types of assemble listing are output.
          ~
-------------------------------------------------------------------------
        .PRINT    LIST,NOSRC,NOCREF
                                        ; Only a section information listing is output.
          ~
```

.LIST

Source Program Listing Output Control

Syntax

```
.LISTΔ<output specifier>[,<output specifier>...]

 Output specifier:  {ON|OFF|COND|NOCOND|DEF|NODEF|CALL|NOCALL|
                     EXP|NOEXP|CODE|NOCODE}
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .LIST mnemonic in the operation field.

3. Operands

   Enter the output specifiers in the operand filed.

Description

1. .LIST is the assembler directive that controls output of the source program listing in the following three ways:

   I Selects whether or not to output source statements.
   II Selects whether or not to output source statements related to the conditional assembly and macro functions.
   III Selects whether or not to output object code lines.

2. Output is controlled by output specifiers as follows:

Type	Output Specifier: Output	Output Specifier: Not output	Object	Description
I	ON	OFF	Source statements	The source statements following this directive
II	COND	NOCOND	Failed condition	Condition-failed .AIF directive statements
	DEF	NODEF	Definition	Macro definition statements .AREPEAT and .AWHILE definition statements .INCLUDE directive statements .ASSIGNA and .ASSSIGNC directive statements
	CALL	NOCALL	Call	Macro call statements, .AIF and .AENDI directive statements
	EXP	NOEXP	Expansion	Macro expansion statements .AREPEAT and .AWHILE expansion statements
III	CODE	NOCODE	Object code lines	The object code lines exceeding the source statement lines

The shaded sections indicate the default settings when the specifier is omitted.

3. The specification of the .LIST directive is only valid when an assemble listing is output.

   References: Source program listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST SOURCE NOSOURCE

4. The assembler gives priority to command line option specifications concerning source program listing output.

   References: Output on the source program listing
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options" SHOW NOSHOW

5. .LIST directive statements themselves are not output on the source program listing.

.LIST

Coding Example

```
        .LIST   NOCOND,NODEF          -------- This statement controls source program
        .MACRO  SHLRN   COUNT,Rd      -------┐ listing output.
SHIFT   .ASSIGNA \COUNT                      |
        .AIF \&SHIFT GE 16                   |
        SHLR16  \Rd                          |
SHIFT   .ASSIGNA \&SHIFT-16                  |
        .AENDI                               |
        .AIF \&SHIFT GE 8                    |
        SHLR8   \Rd                          |
SHIFT   .ASSIGNA \&SHIFT-8                   |
        .AENDI                               |
        .AIF \&SHIFT GE 4                    |  These statements define a general-
        SHLR2   \Rd                          |  purpose multiple-bit shift procedure as a
        SHLR2   \Rd                          |  macro instruction.
SHIFT   .ASSIGNA \&SHIFT-4                   |
        .AENDI                               |
        .AIF \&SHIFT GE 2                    |
        SHLR2   \Rd                          |
SHIFT   .ASSIGNA \&SHIFT-2                   |
        .AENDI                               |
        .AIF \&SHIFT GE 1                    |
        SHLR    \Rd                          |
        .AENDI                               |
        .ENDM                         -------┘
        SHLRN   23,R0                 -------- Macro call
        .END
```

Note: This example and its description assume that no command line options concerning source program listing output are specified.

Source Listing Output of Coding Example

The .LIST assembler directive suppresses the output of the macro definition, .ASSIGNA directive statement, and .AIF condition-failed statements.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***     07/09/93 16:33:49
                                             PAGE     1
PROGRAM NAME =

   31                                31
   32                                32            SHLRN   23,R0
   33                                   M
   35                                   M
   36                                   M          .AIF 23 GE 16
   37 00000000 4029                     C          SHLR16  R0
   39                                   M          .AENDI
   40                                   M
   41                                   M          .AIF 7 GE 8
   45                                   M
   46                                   M          .AIF 7 GE 4
   47 00000002 4009                     C          SHLR2   R0
   48 00000004 4009                     C          SHLR2   R0
   50                                   M          .AENDI
   51                                   M
   52                                   M          .AIF 3 GE 2
   53 00000006 4009                     C          SHLR2   R0
   55                                   M          .AENDI
   56                                   M
   57                                   M          .AIF 1 GE 1
   58 00000008 4001                     C          SHLR    R0
   59                                   M          .AENDI
   60                                33            .END
 *****TOTAL ERRORS       0
 *****TOTAL WARNINGS     0
```

.FORM

Assemble Listing Line Count and Column Count Setting

Syntax

```
.FORMΔ<size specifier>[,<size specifier>...]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .FORM mnemonic in the operation field.

3. Operands: size specifier

Size Specifier	Listing Size
LIN=<line count>	The specified value is set to the number of lines per page.
COL=<column count>	The specified value is set to the number of columns per line.

   These specifications determine the number of lines and columns in the assemble listing.

Description

1. .FORM is the assembler directive that sets the number of lines per page and columns per line in the assemble listing.

2. The line count and column count must be specified as follows:

   - The specifications must be absolute values,
     and,
   - Forward reference symbols must not appear in the specifications.

   The values allowed for the line count are from 20 to 255.

   The values allowed for the column count are from 79 to 255.

3. The .FORM directive can be used any number of times in a given source program.

4. The assembler gives priority to command line option specifications concerning the number of lines and columns in the assemble listing.

   References: Setting the line count in assemble listing
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options" LINES

   Setting the column count in assemble listing
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options" COLUMNS

5. When there is no specification of command line option or .FORM assembler directive specification for the line count or the column count, the following values are used:

   - Line count..... 60 lines
   - Column count..... 132 columns

Coding Example

Note: This example and its description assume that no command line options concerning the assemble listing line count and/or column count are specified.

```
~

.FORM    LIN=60, COL=200    ; Starting with this page, the number of lines
                            ; per page in the assemble listing is 60 lines.
                            ; Also, starting with this line, the number of
                            ; columns per line in the assemble listing is
                            ; 200 columns.
~

.FORM    LIN=55, COL=150    ; Starting with this page, the number of lines
                            ; per page in the assemble listing is 55 lines.
                            ; Also, starting with this line, the number of
                            ; columns per line in the assemble listing is
                            ; 150 columns.
~
```

`.HEADING`

Source Program Listing Header Setting

Syntax

```
.HEADINGΔ"<character string>"
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .HEADING mnemonic in the operation field.

3. Operands: character string

   Enter the header for the source program listing in the operand field.

Description

1. .HEADING is the assembler directive that sets the header for the source program listing.

   A character string of up to 60 characters can be specified as the header.

   References: Character strings
   → Programmer's Guide, 1.7, "Character Strings"

2. The .HEADING directive can be used any number of times in a given source program.

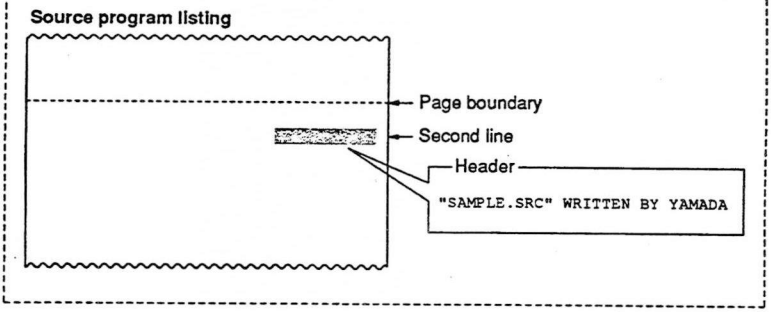
   The range of validity for a given use of the .HEADING directive is as follows:

   - When the .HEADING directive is on the first line of a page, it is valid starting with that page.
   - When the .HEADING directive appears on the second or later line of a page, it is valid starting with the next page.

Coding Example

```
~
.HEADING  """SAMPLE.SRC""  WRITTEN BY YAMADA"
~
```

Explanatory Figure for the Coding Example



.PAGE

Source Program Listing New Page Insertion

Syntax

```
.PAGE
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PAGE mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .PAGE is the assembler directive that inserts a new page in the source program listing at an arbitrary point.
2. The .PAGE directive is ignored if it is used on the first line of a page.
3. .PAGE directive statements themselves are not output to the source program listing.

Coding Example

```
        ~

        MOV     R0,R1
        RTS
        MOV     R0,R2
        .PAGE                           ; A new page is specified here since the
                                        ; section changes at this point.
        .SECTION  DT,DATA,ALIGN=4
        .DATA.L   H'11111111
        .DATA.L   H'22222222
        .DATA.L   H'33333333

        ~
```

Explanatory Figure for the Coding Example

Source program listing

```
    18  00000022  6103              18          MOV       R0,R1
    19  00000024  000B              19          RTS
    20  00000026  6203              20          MOV       R0,R2
```

← New page

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***      10/10/93 10:23:30
PROGRAM NAME =

    22  00000000                    22          .SECTION  DT,DATA,ALIGN
    23  00000000  11111111          23          .DATA.L   H'11111111
    24  00000004  22222222          24          .DATA.L   H'22222222
    25  00000008  33333333          25          .DATA.L   H'33333333
```

Note: See appendix C, "Assemble Listing Output Example", for an explanation of the contents of the source program listing.

.SPACE

Source Program Listing Blank Line Output

Syntax

```
.SPACE[Δ<line count>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .SPACE mnemonic in the operation field.

3. Operands: line count

   Enter the number of blank lines in the operand field.

   A single blank line is output if this operand is omitted.

Description

1. .SPACE is the assembler directive that outputs the specified number of blank lines to the source program listing. Nothing is output for the lines output by the .SPACE directive; in particular line numbers are not output for these lines.

2. The line count must be specified as follows:

   - The specification must be an absolute value,
     and,
   - Forward reference symbols must not appear in the specification.

   Values from 1 to 50 can be specified as the line count.

3. When a new page occurs as the result of blank lines output by the .SPACE directive, any remaining blank lines are not output on the new page.

4. .SPACE directive statements themselves are not output to the source program listing.

Coding Example

```
        .SECTION    DT1,DATA,ALIGN=4
        .DATA.L     H'11111111
        .DATA.L     H'22222222
        .DATA.L     H'33333333
        .DATA.L     H'44444444              ; Inserts five blank lines at the point
        .SPACE  5                           ; where the section changes.
        .SECTION    DT2,DATA,ALIGN=4
            ~
```

Explanatory Figure for the Coding Example

Source program listing

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***      10/10/93 10:23:30
PROGRAM NAME =

    1  00000000                          1           .SECTION   DT1,DATA,ALIGN=4
    2  00000000  11111111                2           .DATA.L    H'11111111
    3  00000004  22222222                3           .DATA.L    H'22222222
    4  00000008  33333333                4           .DATA.L    H'33333333
    5  0000000C  44444444                5           .DATA.L    H'44444444




    6  00000000                          6           .SECTION   DT2,DATA,ALIGN=4
            ~
```

Note: See appendix C, "Assemble Listing Output Example", for an explanation of the contents of the source program listing.

4.2.7 Other Assembler Directives

This assembler provides the following additional assembler directives.

.PROGRAM

Sets the name of the object module.

.RADIX

Sets the radix in which integer constants with no radix specifier are interpreted.

.END

Declares the end of the source program.

Object Module Name Setting

Syntax

```
.PROGRAMΔ<object module name>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .PROGRAM mnemonic in the operation field.

3. Operands: <object module name>

   Enter a name that identifies the object module in the operand field.

Description

1. .PROGRAM is the assembler directive that sets the object module name.

   The object module name is a name that is required by the H Series Linkage Editor or the H Series Librarian to identify the object module.

2. Object module naming conventions are the same as symbol naming conventions.

   The assembler distinguishes upper-case and lower-case letter in object module names.

   References: Coding of symbols
   → Programmer's Guide, 1.3.2, "Coding of Symbols"

3. Setting the object module name with the .PROGRAM directive is valid only once in a given program. (The assembler ignores the second and later specifications of the .PROGRAM directive.)

4. If there is no .PROGRAM specification of the object module name, the assembler will set a default (implicit) object module name.

   The default object module name is the file name of the object file (the object module output destination).

   Example: Object file name ····· PROG . OBJ

   File name File format

   ↓

   Object module name ····· PROG

   References: User's Guide, 1.2, "File Specification Format"

5. The object module name can be the same as a symbol used in the program.

Coding Example

```
.PROGRAM   PROG1      ; This statement sets the object module name to be
                      ; PROG1.
   ~
```

Default Integer Constant Radix Setting

Syntax

```
.RADIXΔ<radix specifier>
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .RADIX mnemonic in the operation field.

3. Operands: radix specifier

Radix Specifier	Radix of Integer Constants with No Radix Specification
B	Binary
Q	Octal
D	Decimal
H	Hexadecimal

The shaded section indicates the default setting when the specifier is omitted.

This specifier sets the radix (base) for integer constants with no radix specification.

Description

1. .RADIX is the assembler directive that sets the radix (base) for integer constants with no radix specification.

2. When there is no radix specification with the .RADIX directive in a program, integer constants with no radix specification are interpreted as decimal numbers.

3. If hexadecimal (radix specifier H) is specified as the radix for integer constants with no radix specification, integer constants whose first digit is A through F must be prefixed with a 0 (zero). (The assembler interprets expressions that begin with A through F to be symbols.)

4. Specifications with the .RADIX directive are valid from the point of specification forward in the program.

Coding Example

```
            ~

            .RADIX  D
X:          .EQU    100         ; This 100 is decimal.

            ~

            .RADIX  H
Y:          .EQU    64          ; This 64 is hexadecimal.

            ~
```

```
            ~

            .RADIX  H
Z:          .EQU    0F          ; A zero is prefixed to this constant "0F" since it would
                                ; be interpreted as a symbol if it were written as simply
                                ; "F".

            ~
```

Source Program End Declaration

Syntax

```
.END[Δ<start address>]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .END mnemonic in the operation field.

3. Operands: start address

   Enter the start address for simulation in the operand field if required.

Description

1. .END is the assembler directive that declares the end of the source program.

   Assembly processing terminates at the point that the .END directive appears.

2. If a start address is specified with the .END directive in the operand field, the simulator/debugger starts simulation from that address.

3. The start address must be specified with either an absolute value or an address value.

4. The value of the start address must be an address in a code section.

```
.END
```

Coding Example

```
        .SECTION    CD,CODE,ALIGN=4
START:
            ~

        .END        START           ; This statement declares the end of the source
                                    ; program.

; The simulator/debugger starts simulation from the address indicated by the value of the
; symbol START.
```

Section 5 File Inclusion Function

The file inclusion function allows source files to be inserted into other source files at assembly time. The file inserted into another file is called an included file.

This assembler provides the .INCLUDE directive to perform file inclusion. The file specified with the .INCLUDE directive is inserted at the location of the .INCLUDE directive.

Example:

Source program

```
.INCLUDE  "FILE.H"

.SECTION CD1,CODE,ALIGN=4
MOV #ON,R0

  ~
```

Included file FILE.H

```
ON:   .EQU  1
OFF:  .EQU  0
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

File included result (source list)

```
      .INCLUDE  "FILE.H"
ON:   .EQU  1
OFF:  .EQU  0

      .SECTION CD1,CODE,ALIGN=4
      MOV #ON,R0

        ~
```

.INCLUDE

File Inclusion

Syntax

```
.INCLUDE␣"<file name>"
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .INCLUDE mnemonic in the operation field.

3. Operands

   Enter the file to be included.

Description

1. .INCLUDE is the file inclusion assembler directive.

2. If no file format is specified, only the file name is used as specified (the assembler does not assume any default file format).

   Reference: User's Guide, 1.2, "File Specification Format"

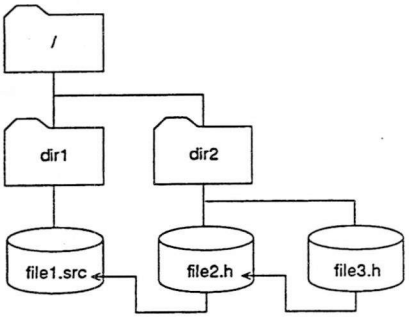
3. The file name can include the directory. The directory can be specified either by the absolute path (path from the route directory) or by the relative path (path from the current directory).

Note: The current directory for the .INCLUDE directive in a source file is the directory where the assembler is initiated. The current directory for the .INCLUDE directive in an included file is the directory where the included file exits.

4. Included files can include other files. The nesting depth for file inclusion is limited to eight levels (multiplex state).

Coding Example

This example assumes the following directory configuration and operations:



- Starts the assembler from the route directory (/)
- Inputs source file /dir1/file1.src
- Inserts file2.h in file1.src
- Inserts file3.h in file2.h

The start command is as follows:

```
>asmsh /dir1/file1.src (RET)
```

file1.src must have the following inclusion directive:

```
   .INCLUDE "dir2/file2.h"     ; / is the current directory (relative path specification).
or
   .INCLUDE "/dir2/file2.h"    ; Absolute path specification
```

file2.h must have the following inclusion directive:

```
   .INCLUDE "file3.h"          ; /dir2 is the current directory (relative path
                                 specification).
or
   .INCLUDE "/dir2/file3.h"    ; Absolute path specification
```

CAUTION!

When using MS-DOS, change the slash in the above example to a backslash (\).

HITACHI

Section 6 Conditional Assembly Function

6.1 Overview of the Conditional Assembly Function

The conditional assembly function provides the following assembly operations:

- Selects whether or not to assemble a specified part of a source program according to the specified condition.
- Iteratively assembles a specified part of a source program.

6.1.1 Preprocessor variables

Preprocessor variables are used to write assembly conditions. Preprocessor variables are of either integer or character type.

1. Integer preprocessor variables

   Integer preprocessor variables are defined by the .ASSIGNA directive (these variables can be redefined).

   When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

Example:

```
FLAG:   .ASSIGNA 1
        .AIF \&FLAG EQ 1        ;  MOV R0,R1 is assembled
         MOV R0,R1              ;  when FLAG is 1.
        .AENDI
```

2. Character preprocessor variables

   Character preprocessor variables are defined by the .ASSIGNC directive (these variables can be redefined).

   When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

Example:

```
FLAG:   .ASSIGNC "ON"
        .AIF "\&FLAG" EQ "ON"   ;  MOV R0,R1 is assembled
         MOV R0,R1              ;  when FLAG is "ON".
        .AENDI
```

6.1.2 Conditional Assembly

The conditional assembly function determines whether or not to assemble a specified part of a source program according to the specified condition. A coding example is shown below.

```
 ~
.AIF  <condition>
  <Statements to be assembled when the condition is satisfied>
.AELSE
  <Statements to be assembled when the condition is not satisfied>
.AENDI
 ~
```

This part can be omitted from the coding.

Example:

```
 ~
.AIF  "\&FLAG" EQ "ON"
 MOV R0,R10              ;  These statements
 MOV R1,R11              ;  will be assembled
 MOV R2,R12              ;  when FLAG is "ON".
.AELSE
 MOV R10,R0              ;  These statements
 MOV R11,R1              ;  will be assembled
 MOV R12,R2              ;  when FLAG is not "ON".
.AENDI
 ~
```

6.1.3 Iterated Expansion

A part of a source program can be iteratively assembled the specified number of times. A coding example is shown below.

```
        ~
.AREPEAT <count>
   <Statements to be iterated>
.AENDR
        ~
```

Example:

```
                        ;  This example is a division of 64-bit data by 32-bit data.
                        ;  R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
TST     R0,R0           ;  Zero divisor check
BT      zero_div
CMP/HS  R0,R1           ;  Overflow check
BT      over_div
DIV0U                   ;  Flag initialization
.AREPEAT 32
ROTCL   R2              ;  These statements are
DIV1    R0,R1           ;  iterated 32 times.
.AENDR
ROTCL   R2              ;  R2 = quotient
```

6.1.4 Conditional Iterated Expansion

A part of a source program can be iteratively assembled while the specified condition is satisfied. A coding example is shown below.

```
        ~
.AWHILE <condition>
    <Statements to be iterated>
.AENDW
        ~
```

Example:

```
                                    ; This example is a multiply and accumulate
                                    ; operation.
TblSiz: .ASSIGNA  50                ; TblSiz: Data table size
        MOV       A_Tbl1,R1         ; R1: Start address of data table 1
        MOV       A_Tbl2,R2         ; R2: Start address of data table 2
        CLRMAC                      ; MAC register initialization
        .AWHILE   \&TblSize GT 0    ; While TblSiz is larger than 0,
        MAC.W     @R0+,@R1+         ; this statement is iteratively assembled.
TblSiz: .ASSIGNA  \&TblSiz-1        ; 1 is subtracted from TblSiz.
        .AENDW
        STS       MACL,R0           ; The result is obtained in R0.
```

6.2 Conditional Assembly Directives

This assembler provides the following conditional assembly directives.

.ASSIGNA	Defines an integer preprocessor variable. The defined variable can be redefined.
.ASSIGNC	Defines a character preprocessor variable. The defined variable can be redefined.
.AIF .AELSE .AENDI	Determines whether or not to assemble a part of a source program according to the specified condition. When the condition is satisfied, the statements after the .AIF are assembled. When not satisfied, the statements after the .AELSE are assembled.
.AREPEAT .AENDR	Repeats assembly of a part of a source program (between .AREPEAT and .AENDR) the specified number of times.
.AWHILE .AENDW	Assembles a part of a source program (between .AWHILE and .AENDW) iteratively while the specified condition is satisfied.
.EXITM	Terminates .AREPEAT or .AWHILE iterated expansion.

.ASSIGNA

Integer Preprocessor Variable Definition (redefinition is possible)

Syntax

```
<preprocessor variable>[:].ASSIGNAΔ<value>
```

Statement Elements

1. Label

   Enter the name of the preprocessor variable.

2. Operation

   Enter the .ASSIGNA mnemonic in the operation field.

3. Operands

   Enter the value to be assigned to the preprocessor variable.

Description

1. .ASSIGNA is the assembler directive that defines a value for an integer preprocessor variable. The syntax of integer preprocessor variables is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

2. The preprocessor variables defined with the .ASSIGNA directive can be redefined with the .ASSIGNA directive.

3. The values for the preprocessor variables must be the following:

   - Constant (integer constant and character constant)
   - Defined preprocessor variable
   - Expression using the above as terms

4. Defined preprocessor variables are valid from the point of specification forward in the source program.

5. Defined preprocessor variables can be referenced in the following locations:

- .ASSIGNA directive
- .ASSIGNC directive
- .AIF directive
- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing integer preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

```
\&<preprocessor variable>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

Coding Example

```
                                ; This example generates a general-purpose multiple-bit
                                ; shift instruction which shifts bits to the right by the
                                ; number of SHIFT.
RN:     .REG      (R0)          ; R0 is set to Rn.
SHIFT:  .ASSIGNA  27            ; 27 is set to SHIFT
        .AIF  \&SHIFT GE 16     ; Condition: SHIFT ≥ 16
        SHLR16  Rn              ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
SHIFT:  .ASSIGNA  \&SHIFT-16    ; 16 is subtracted from SHIFT.
        .AENDI
        .AIF  \&SHIFT GE 8      ; Condition: SHIFT ≥ 8
        SHLR8   Rn              ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
SHIFT:  .ASSIGNA  \&SHIFT-8     ; 8 is subtracted from SHIFT.
        .AENDI
        .AIF  \&SHIFT GE 4      ; Condition: SHIFT ≥ 4
        SHLR2   Rn              ; When the condition is satisfied, Rn is shifted to the right by 4 bits.
        SHLR2   Rn              ;
SHIFT:  .ASSIGNA  \&SHIFT-4     ; 4 is subtracted from SHIFT.
        .AENDI
        .AIF  \&SHIFT GE 2      ; Condition: SHIFT ≥ 2
        SHLR2   Rn              ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
SHIFT:  .ASSIGNA  \&SHIFT-2     ; 2 is subtracted from SHIFT.
        .AENDI
        .AIF  \&SHIFT EQ 1      ; Condition: SHIFT = 1
        SHLR    Rn              ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
        .AENDI
```

The expanded results are as follows:

```
        SHLR16  R1      ; When the condition is satisfied, Rn is shifted to the right by 16 bits.
        SHLR8   R1      ; When the condition is satisfied, Rn is shifted to the right by 8 bits.
        SHLR2   R1      ; When the condition is satisfied, Rn is shifted to the right by 2 bits.
        SHLR1   R1      ; When the condition is satisfied, Rn is shifted to the right by 1 bit.
```

Character Preprocessor Variable Definition (redefinition is possible)

Syntax

```
<preprocessor variable>[:].ASSIGNCΔ"<character string>"
```

Statement Elements

1. Label

   Enter the name of the preprocessor variable.

2. Operation

   Enter the .ASSIGNC mnemonic in the operation field.

3. Operands

   Enter the character string enclosed with double quotation marks (").

Description

1. .ASSIGNC is the assembler directive that defines a character string for an character preprocessor variable. The syntax of character preprocessor variables is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

2. The preprocessor variables defined with the .ASSIGNC directive can be redefined with the .ASSIGNC directive.

3. Character strings are specified by characters or preprocessor variables enclosed by double quotation marks (").

4. Defined preprocessor variables are valid from the point of specification forward in the source program.

5. Defined preprocessor variables can be referenced in the following locations:

   - .ASSIGNA directive
   - .ASSIGNC directive
   - .AIF directive

- .AREPEAT directive
- .AWHILE directive
- Macro body (source statements between .MACRO and .ENDM)

When referencing character preprocessor variables, insert a backslash (\) and an ampersand (&) in front of them.

```
\&<preprocessor variable>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

Coding Example

```
FLAG:   .ASSIGNC  "ON"          ; "ON" is set to FLAG.
        ~
        .AIF  "\&FLAG" EQ "ON"  ; MOV R0,R1 is assembled
        MOV R0,R1               ; when FLAG is "ON".
        .AENDI
        ~
FLAG:   .ASSIGNC  "\&FLAG "     ; A space (" ") is added to FLAG.
FLAGA:  .ASSIGNC  "OFF"         ; "OFF" is added to FLAGA.
FLAG:   .ASSIGNC  "\&FLAG'AND \&FLAGA"
                                ; An apostrophe (') is used to distinguish FLAG and AND.
                                ; FLAG finally becomes "ON AND OFF".
        ~
```

.AIF	.AELSE	.AENDI

Conditional Assembly

Syntax

```
.AIFΔ<term1>Δ<relational operator>Δ<term2>
<Source statements assembled if the condition is satisfied>
[.AELSE
<Source statements assembled if the condition is not satisfied>]
.AENDI
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AIF, .AELSE (can be omitted), or .AENDI mnemonic in the operation field.

3. Operands

   .AIF: Enter the condition. Refer to the description below.
   .AELSE: The operand field is not used.
   .AENDI: The operand field is not used.

Description

1. .AIF, .AELSE, and .AENDI are the assembler directives that select whether or not to assemble source statements according to the condition specified. The .AELSE directive can be omitted.

2. The condition must be specified as follows:

   `.AIFΔ<term1>Δ<relational operator>Δ<term2>`

Terms are specified with numeric values or character strings. However, when a numeric value and a character string are compared, the condition always fails.

Numeric values are specified by constants or preprocessor variables.

Character strings are specified by characters or preprocessor variables enclosed by double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks (" ") in succession.

3. The following relational operators can be used:

EQ: term1 = term2
NE: term1 ≠ term2
GT: term1 > term2
LT: term1 < term2
GE: term1 ≥ term2
LE: term1 ≤ term2

Note: Numeric values are handled as 32-bit signed integers. For character strings, only EQ and NE conditions can be used.

Coding Example

```
        ~
.AIF "\&FLAG" EQ "ON"
 MOV R0,R10             ; These statements
 MOV R1,R11             ; are assembled
 MOV R2,R12             ; when FLAG is "ON".
.AELSE
 MOV R10,R0             ; These statements
 MOV R11,R1             ; are assembled
 MOV R12,R2             ; when FLAG is not "ON".
.AENDI
        ~
```

.AREPEAT	.AENDR

Iterated Expansion

Syntax

```
.AREPEAT <count>
<Source statements iteratively assembled>
.AENDR
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AREPEAT or .AENDR mnemonic in the operation field.

3. Operands

   .AREPEAT: Enter the number of iterations.
   .AENDR: The operand field is not used.

Description

1. .AREPEAT and .AENDR are the assembler directives that assemble source statements by iteratively expanding them the specified number of times.

2. The source statements between the .AREPEAT and .AENDR directives are iterated the number of times specified with the .AREPEAT directive. Note that the source statements are simply copied the specified number of times, and therefore, the operation does not loop at program execution.

3. Counts are specified by constants or preprocessor variables.

4. Nothing is expanded if a value of 0 or smaller is specified.

.AREPEAT	.AENDR

Coding Example

```
                                ;  This example is a division of 64-bit data by 32-bit data.
                                ;  R1:R2 (64 bits) ÷ R0 (32 bits) = R2 (32 bits): Unsigned
TST     R0,R0                   ;  Zero divisor check
BT      zero_div
CMP/HS  R0,R1                   ;  Overflow check
BT      over_div
DIV0U                           ;  Flag initialization
.AREPEAT 32
ROTCL   R2                      ;  These statements are
DIV1    R0,R1                   ;  iterated 32 times.
.AENDR
ROTCL   R2                      ;  R2 = quotient
```

.AWHILE	.AENDW

Conditional Iterated Expansion

Syntax

```
.AWHILEΔ<term1>Δ<relational operator>Δ<term2>
<Source statements iteratively assembled>
.AENDW
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .AWHILE or .AENDW mnemonic in the operation field.

3. Operands

   .AWHILE: Enter the condition to iteratively expand source statements.
   .AENDW: The operand field is not used.

Description

1. .AWHILE and .AENDW are the assembler directives that assemble source statements by iteratively expanding them while the specified condition is satisfied.

2. The source statements between the .AWHILE and .AENDW directives are iterated while the condition specified with the .AWHILE directive is satisfied. Note that the source statements are simply copied iteratively, and therefore, the operation does not loop at program execution.

3. The condition must be specified as follows:

   ```
   .AWHILEΔ<term1>Δ<relational operator>Δ<term2>
   ```

   Terms are specified with numeric values or character strings. However, when a numeric value and a character string are compared, the condition always fails.

Numeric values are specified by constants or preprocessor variables.

Character strings are specified by characters or preprocessor variables enclosed by double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks (" ") in succession.

Conditional iterated expansion terminates when the condition finally fails. An infinite loop occurs if a condition which never fails is specified. Accordingly, the condition for this directive must be carefully specified.

4. The following relational operators can be used:

EQ: term1 = term2
NE: term1 ≠ term2
GT: term1 > term2
LT: term1 < term2
GE: term1 ≥ term2
LE: term1 ≤ term2

Note: Numeric values are handled as 32-bit signed integers. For character strings, only EQ and NE conditions can be used.

Coding Example

```
                                    ; This example is a multiply and accumulate
                                    ; operation.
TblSiz:  .ASSIGNA  50               ; TblSiz: Data table size
         MOV       A_Tbl1,R1        ; R1: Start address of data table 1
         MOV       A_Tbl2,R2        ; R2: Start address of data table 2
         CLRMAC                     ; MAC register initialization
         .AWHILE   \&TblSize GT 0   ; While TblSiz is larger than 0,
         MAC.W     @R0+,@R1+        ; this statement is iteratively assembled.
TblSiz:  .ASSIGNA  \&TblSiz-1       ; 1 is subtracted from TblSiz.
         .AENDW
         STS       MACL,R0          ; The result is obtained in R0.
```

Expansion Termination

Syntax

```
.EXITM
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXITM mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .EXITM is the assembler directive that terminates an iterated expansion (.AREPEAT to .AENDR) or a conditional iterated expansion (.AWHILE to .AENDW).

2. Either expansion is terminated when this directive appears.

3. This directive is also used to exit from macro expansions. The location of this directive must be specified carefully when macro instructions and iterated expansion are combined.

   Reference: Macro expansion
   → Programmer's Guide, 7.2, "Macro Function Directives"

Coding Example

```
          ~
COUNT     .ASSIGNA 0              ; 0 is set to COUNT.
          .AWHILE 1 EQ 1          ; An infinite loop (condition is always satisfied) is
                                  ; specified.
          ADD      R0,R1
          ADD      R2,R3

COUNT     .ASSIGNA \&COUNT+1      ; 1 is added to COUNT.
          .AIF     \&COUNT EQ 2 ; Condition: COUNT = 2
          .EXITM
          .AENDI
          .AENDW
          ~
```

When COUNT is updated and satisfies the condition specified with the .AIF directive, .EXITM is assembled. When .EXITM is assembled, .AWHILE expansion is terminated.

The expansion results are as follows:

```
ADD  R0,R1 ····· When COUNT is 0
ADD  R2,R3
ADD  R0,R1 ····· When COUNT is 1
ADD  R2,R3
```

After this, COUNT becomes 2 and expansion is terminated.

Section 7 Macro Function

7.1 Overview of the Macro Function

The macro function allows commonly used sequences of instructions to be named and defined as one macro instruction. This is called a macro definition. Macro instructions are defined as follows:

```
~
.MACRO <macro name>
   <macro body>
.ENDM
~
```

A macro name is the name assigned to a macro instruction, and a macro body is the statements to be executed as the macro instruction.

Using a defined macro instruction by specifying the name is called a macro call. Macro instructions are called as follows:

```
~
<defined macro name>
~
```

An example of macro definition and macro call is shown below.

Example:

```
~
.MACRO  SUM          ; Processing to obtain the sum of R0, R1, R2,
 MOV  R0,R10         ; and R3 is defined as macro instruction SUM.
 ADD  R1,R10
 ADD  R2,R10
 ADD  R3,R10
.ENDM
~
SUM                  ; This statement calls macro instruction SUM.
                     ; Macro body  MOV  R0,R10
                     ;             ADD  R1,R10
                     ;             ADD  R2,R10
                     ;             ADD  R3,R10
                     ; is expanded from the macro instruction.
```

Parts of the macro body can be replaced when expanded by the following procedure:

1. Macro definition

   a. Declare formal parameters in the .MACRO directive.

   b. Use the formal parameters in the macro body. Formal parameters must be identified in the macro body by placing a backslash (\) in front of them.

2. Macro call

   Specify macro parameters in the macro call.

   When the macro instruction is expanded, the formal parameters are replaced with their corresponding macro parameters.

Example:

```
~
.MACRO  SUM  ARG1          ; Formal parameter ARG1 is defined.
 MOV R0, \ARG1             ; ARG1 is referenced in the macro body.
 ADD R1, \ARG1
 ADD R2, \ARG1
 ADD R3, \ARG1
.ENDM
~

 SUM  R10                  ; This statement calls macro instruction SUM
                           ; specifying macro parameter R10.
                           ; The formal parameter in the macro body is
                           ; replaced with the macro parameter, and
                           ;         MOV  R0,R10
                           ;         ADD  R1,R10
                           ;         ADD  R2,R10
                           ;         ADD  R3,R10 is expanded.
```

7.2 Macro Function Directives

This assembler provides the following macro function directives.

.MACRO .ENDM	Defines a macro instruction.
.EXITM	Terminates macro instruction expansion.

.MACRO	.ENDM

Macro Definition

Syntax

```
.MACROΔ<macro name>[Δ<formal parameter>[=<default>]
                                    [,<formal parameter>...]]
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .MACRO or .ENDM mnemonic in the operation field.

3. Operands

   .MACRO: Enter the name and formal parameters for the macro instruction to be defined. When formal parameters are defined, their defaults can be defined (defaults can be omitted).
   .ENDM: The operand filed is not used.

Description

1. .MACRO and .ENDM are the assembler directives that define a macro instruction (a sequence of source statements that are collectively named and handled together).

2. Macro definition

   Naming as a macro instruction the source statements (macro body) between the .MACRO and .ENDM directives is called a macro definition.

3. Macro name

   Macro names are the names assigned to macro instructions.

4. Formal parameters

Formal parameters are specified so that parts of the macro body can be replaced by specific

parameters at expansion time. Formal parameters are replaced with the character strings (macro parameters) specified at macro expansion (macro call).

a. Formal parameter syntax

The syntax for formal parameters is the same as that for symbols. The assembler distinguishes uppercase and lowercase letters.

b. Formal parameter reference

Formal parameters are used (referenced) at the part to be replaced in the the macro body.

The syntax of formal parameter reference in macro bodies is as follows:

```
\<formal parameter name>[']
```

To clearly distinguish the preprocessor variable name from the rest of the source statement, an apostrophe (') can be added.

3. Formal parameter defaults

Defaults for formal parameters can be specified in macro definitions. The default specifies the character string to replace the formal parameter when the corresponding macro parameter is omitted in a macro call.

The default must be enclosed by double quotation marks (") or angle brackets (<>) if any of the following characters are included in the default.

- Space
- Tab
- Comma (,)
- Semicolon (;)
- Double quotation marks (")
- Angle brackets (< >)

The assembler inserts defaults at macro expansion by removing the double quotation marks or angle brackets that enclose the character strings.

.MACRO	.ENDM

6. Restrictions on macro definitions

   a. Macros cannot be defined in the following locations:

      - Macro bodies (between .MACRO and .ENDM directives)
      - Between .AREPEAT and .AENDR directives
      - Between .AWHILE and .AENDW directives

   b. The .ENDM directive cannot be used within a macro body.

   c. No symbol can be inserted in the label field of the .ENDM directive. The .ENDM directive is ignored if its label field is not blank, but no error is generated in this case.

Coding Example

```
~
MACRO  SUM              ; Processing to obtain the sum of R0, R1, R2,
 MOV R0,R10             ; and R3 is defined as macro instruction SUM.
 ADD R1,R10
 ADD R2,R10
 ADD R3,R10
ENDM
~
SUM                     ; This statement calls macro instruction SUM
                        ; Macro body  MOV  R0,R10
                        ;             ADD  R1,R10
                        ;             ADD  R2,R10
                        ;             ADD  R3,R10 is expanded.
```

Expansion Termination

Syntax

```
.EXITM
```

Statement Elements

1. Label

   The label field is not used.

2. Operation

   Enter the .EXITM mnemonic in the operation field.

3. Operands

   The operand field is not used.

Description

1. .EXITM is the assembler directive that terminates a macro expansion. This directive can be specified within the macro body (between the .MACRO and .ENDM directives).

2. Expansion is terminated when this directive appears.

3. This directive is also used to exit from iterated expansions specified with the .AREPEAT or .AWHILE directive. The location of this directive must be specified carefully when macro instructions and iterated expansion are combined.

.EXITM

Coding Example

```
        .MACRO  SUM P1
        MOV     R0,R10          ┐
        ADD     R1,R10          ┘ (1)
        ADD     R2,R10
        \P1                     -------- (2)
        ADD     R3,R10
        .ENDM

        ~

        SUM     .EXITM
```

.EXITM is expanded at (2) and macro expansion is terminated. Only the statements indicated by (1) are expanded.

7.3 Macro Body

The source statements between the .MACRO and .ENDM directives are called a macro body. The macro body is expanded and assembled by a macro call.

1. Formal parameter reference

   Formal parameters are used to specify the parts to be replaced with macro parameters at macro expansion.

   The syntax of formal parameter reference in macro bodies is as follows:

   ```
   \<formal parameter name>[']
   ```

   To clearly distinguish the formal parameter name from the rest of the source statement, add an apostrophe (').

Coding example:

```
.MACRO   PLUS1 P,P1      ; P and P1 are formal parameters.
ADD      #1,\P1          ; Formal parameter P1 is referenced.
.SDATA   "\P'1"          ; Formal parameter P is referenced.
.ENDM
PLUS1    R,R1            ; PLUS1 is expanded.
  ~
```

Expanded results are as follows:

```
ADD      #1,R1           ; Formal parameter P1 is referenced.
.SDATA   "R1"            ; Formal parameter P is referenced.
```

2. Preprocessor variable reference

   Preprocessor variables can be referenced in macro bodies.

   The syntax for preprocessor variable reference is as follows:

   ```
   \&<preprocessor variable name>[']
   ```

   To clearly distinguish the formal parameter name from the rest of the source statement, add an apostrophe (').

Coding example:

```
        .MACRO  PLUS1
        ADD     #1,R\&V1        ; Preprocessor variable V1 is referenced.
        .SDATA  "\&V'1"         ; Preprocessor variable V is referenced.
        .ENDM
V       .ASSIGNC "R"            ; Preprocessor variable V is defined.
V1      .ASSIGNA 1              ; Preprocessor variable V1 is defined.
        PLUS1                   ; PLUS1 is expanded.
```

Expanded results are as follows:

```
        ADD     #1,R1           ; Preprocessor variable V1 is referenced.
        .SDATA  "R1"            ; Preprocessor variable V is referenced.
```

3. Macro generation number

   The macro generation number facility is used to avoid the problem that symbols used within a macro body will be multiply defined if the macro is expanded multiple times. To avoid this problem, specify the macro generation number marker as part of any symbol used in a macro. This will result in symbols that are unique to each macro call.

   The macro generation number marker is expanded as a 5-digit decimal number (between 00000 and 99999) unique to the macro expansion.

   The syntax for specifying the macro generation number marker is as follows:

   ```
   \@
   ```

   Two or more macro generation number markers can be written in a macro body, and they will be expanded to the same number in one macro call.

CAUTION!

Because macro generation number markers are expanded to numbers, they must not be written at the beginning of symbol names.

Reference: Programmer's Guide, 1.3.2, "Coding of Symbols"

Coding example:

```
        .MACRO  RES_STR STR, Rn
        MOV.L   #str\@,\Rn
        BRA     end_str\@
        NOP
str\@   .SDATA  "\STR"
        .ALIGN 2
end_str\@
        .ENDM

        RES_STR "ONE",R0
        RES_STR "TWO",R1
```

Different symbols are generated each time RES_STR is expanded.

Expanded results are as follows:

```
        MOV.L   #str00000,R0
        BRA     end_str00000
        NOP
str00000        .SDATA  "ONE"
        .ALIGN 2

end_str00000
        MOV.L   #str00001,R1
        BRA     end_str00001
        NOP
str00001        .SDATA  "TWO"
        .ALIGN 2

end_str00001
```

4. Macro replacement processing exclusion

When a backslash (\) appears in a macro body, it specifies macro replacement processing. Therefore a means for excluding this macro processing is required when it is necessary to use the backslash as an ASCII character.

The syntax for macro replacement processing exclusion is as follows:

```
\(<macro replacement processing excluded character string>)
```

The backslash and the parentheses will be removed in macro processing.

Coding example:

```
.MACRO   BACK_SLASH_SET
\(MOV        #"\",R0)      ; \ is expanded as an ASCII character.
.ENDM
```

Expanded results are as follows:

```
MOV      #"\",R0           ; \ is expanded as an ASCII character.
```

5. Comments in macros

Comments in macro bodies can be coded as normal comments or as macro internal comments. When comments in the macro body are not required in the macro expansion code (to avoid repeating the same comment in the listing file), those comments can be coded as macro internal comments to suppress their expansion.

The syntax for macro internal comments is as follows:

```
\;<comment>
```

Coding example:

```
.MACRO PUSH Rn
MOV.L    \Rn,@-R15       \;   \Rn is a register.
.ENDM
PUSH     R0
```

Expanded results are as follows (the comment is not expanded):

```
MOV.L    R0,@-R15
```

6. Character string manipulation functions

Character string manipulation functions can be used in the body of a macro. The following character string manipulation functions are provided.

.LEN ..... Character string length.
.INSTR ..... Character string search.
.SUBSTR ..... Character string substring.

References:

.LEN → Programmer's Guide, 7.5, "Character String Manipulation Functions", .LEN
.INSTR → Programmer's Guide, 7.5, "Character String Manipulation Functions", .INSTR
.SUBSTR → Programmer's Guide, 7.5, "Character String Manipulation Functions", .SUBSTR

7.4 Macro Call

Expanding a defined macro instruction is called a macro call. The syntax for macro calls is as follows:

Syntax

```
[<symbol>] <macro name>[<macro parameter>
                                    [,<macro parameter> ...]]
```

Statement Elements

1. Label

   Enter a reference symbol in the label field if required.

2. Operation

   Enter the macro name to be expanded in the operation field. The macro name must have been already defined before a macro call.

3. Operands

   Enter character strings as macro parameters to replace formal parameters at macro expansion. The formal parameters must have been declared in the macro definition with .MACRO.

Description

1. Macro parameter specification

   Macro parameters can be specified by either positional specification or keyword specification.

   a. Positional specification

      The macro parameters are specified in the same order as that of the formal parameters declared in the macro definition.

   b. Keyword specification

      Each macro parameter is specified following its corresponding formal parameter, separated by an equal sign (=).

2. Macro parameter syntax

Macro parameters must be enclosed by double quotation marks (") or angle brackets (<>) if any of the following characters are included in the macro parameters:

- Space
- Tab
- Comma (,)
- Semicolon (;)
- Double quotation marks (")
- Angle brackets (< >)

Macro parameters are inserted by removing the double quotation marks or angle brackets that enclose character strings at macro expansion.

Coding Example

```
        .MACRO  SUM FROM=0,TO=9         ; Macro instruction SUM and formal
                                        ; parameters FROM and TO are defined.
        MOV     R\FROM,R10
COUNT   .ASSIGNA        \FROM+1
        .AWHILE \&COUNT LE \TO          Macro body is coded
        MOV     R\&COUNT,R10            using formal parameters.
COUNT   .ASSIGNA        \&COUNT+1
        .AENDW
        .ENDM

        SUM     0,5                     Both will be expanded
        SUM     TO=5                    into the same statements.
```

Expanded results are as follows (the formal parameters in the macro body are replaced with macro parameters):

```
        MOV     R0, R10
        MOV     R1, R10
        MOV     R2, R10
        MOV     R3, R10
        MOV     R4, R10
        MOV     R5, R10
```

7.5 Character String Manipulation Functions

This assembler provides the following character string manipulation functions.

.LEN	Counts the length of a character string.
.INSTR	Searches for a character string.
.SUBSTR	Extracts a character string.

.LEN

Character String Length Count

Syntax

```
.LEN[Δ]("<character string>")
```

Description

1. .LEN counts the number of characters in a character string and replaces itself with the number of characters in decimal with no radix.

2. Character strings are specified by enclosing the desired characters in double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks in succession.

3. Macro formal parameters and preprocessor variables can be specified in the character string as shown below.

```
.LEN("\<formal parameter>")
.LEN("\&<preprocessor variable>")
```

4. This function can only be used within a macro body (between .MACRO and .ENDM directives).

Coding Example:

```
        ~
        .MACRO RESERVE_LENGTH P1
        .ALIGN 4
        .SRES  .LEN("\P1")
        .ENDM
        ~
        RESERVE_LENGTH ABCDEF
        RESERVE_LENGTH ABC
```

Expanded results are as follows:

```
        .ALIGN  4
        .SRES   6               ; "ABCDEF" has six characters.
        .ALIGN  4
        .SRES   3               ; "ABC" has three characters.
```

.INSTR

Character String Search

Syntax

```
.INSTR[Δ]("<character string 1>","<character string 2>"
                                        [,<start position>])
```

Description

1. .INSTR searches character string 1 for character string 2, and replaces itself with the numerical value of the position of the found string (with 0 indicating the start of the string) in decimal with no radix. .INSTR is replaced with –1 if character string 2 does not appear in character string 1.

2. Character strings are specified by enclosing the desired characters in double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks in succession.

3. The <start position> parameter specifies the search start position as a numerical value, with 0 indicating the start of the string. Zero is used as default when this parameter is omitted.

4. Macro formal parameters and preprocessor variables can be specified in the character strings and as the start position as shown below.

   ```
   .INSTR("\<formal parameter>", ...)

   .INSTR("\&<preprocessor variable>", ...)
   ```

5. This function can only be used within a macro body (between the .MACRO and .ENDM directives).

Coding Example:

```
        ~
        .MACRO FIND_STR P1
        .DATA.W .INSTR("ABCDEFG","\P1",0) .
        .ENDM
        ~

        FIND_STR CDE
        FIND_STR H
```

Expanded results are as follows:

```
        .DATA.W  2          ; The start position of "CDE" is 2 (0 indicating the
                            ; beginning of the string) in "ABCDEFG"
        .DATA.W  -1         ; "ABCDEFG" includes no "H".
```

.SUBSTR

Character Substring Extraction

Syntax

```
.SUBSTR[Δ]("<character string>",<start position>
                                        ,<extraction length>)
```

Description

1. .SUBSTR extracts from the specified character string a substring starting at the specified start position of the specified length. .SUBSTR is replaced with the extracted character string enclosed by double quotation marks (").
2. Character strings are specified by enclosing the desired characters in double quotation marks ("). To specify a double quotation mark in a character string, enter two double quotation marks in succession.
3. The value of the extraction start position must be 0 or greater. The value of the extraction length must be 1 or greater.
4. If illegal or inappropriate values are specified for the <start position> or <extraction length> parameters, this function is replaced with a blank space (" ").
5. Macro formal parameters and preprocessor variables can be specified in the character string, and as the start position and extraction length parameters as shown below.

   ```
   .SUBSTR("\<formal parameter>", ...)
   ```

   ```
   .SUBSTR("\&<preprocessor variable>", ...)
   ```

6. This function can only be used within a macro body (between the .MACRO and .ENDM directives).

Coding Example:

```
 ~
.MACRO RESERVE_STR P1=0 P2
.SDATA  .SUBSTR("ABCDEFG",\P1,\P2)
.ENDM
 ~

RESERVE_STR 2,2
RESERVE_STR ,3      ; Macro parameter P1 is omitted.
```

Expanded results are as follows:

```
.SDATA  "CD"
.SDATA  "ABC"
```

HITACHI

Section 8 Automatic Literal Pool Generation Function

8.1 Overview of Automatic Literal Pool Generation

To move 2-byte or 4-byte constant data (referred to below as a "literal") to a register, a literal pool (a collection of literals) must be reserved and referred to in PC relative addressing mode. For literal pool location, the following must be considered:

- Is data stored within the range that can be accessed by data move instructions?
- Is 2-byte data aligned to a 2-byte boundary and is 4-byte data aligned to a 4-byte boundary?
- Can data be shared by several data move instructions?
- Where should the literal pool be located in the program?

The assembler automatically generates from a single instruction a .DATA directive and a PC relative MOV or MOVA instruction, which moves constant data to a register.

For example, this function enables program (a) below to be coded as (b):

(a)

```
        MOV.L   DATA1,R0
        MOV.L   DATA2,R1

        ~

        .ALIGN 4
DATA1   .DATA.L H'12345678
DATA2   .DATA.L 50000
```

(b)

```
        MOV.L   #H'12345678,R0
        MOV.L   #500000,R1

        ~
```

8.2 Extended Instructions Related to Automatic Literal Pool Generation

The assembler automatically generates a literal pool corresponding to an extended instruction (MOV.W #imm, Rn; MOV.L #imm, Rn; or MOVA #imm, R0) and calculates the PC relative displacement value.

An extended instruction source statement is expanded to an executable instruction and literal data as shown in table 8-1.

Table 8-1 Extended Instructions and Expanded Results

Extended Instruction	Expanded Result
MOV.W #imm, Rn	MOV.W @(disp, PC), Rn and 2-byte literal data
MOV.L #imm, Rn	MOV.L @(disp, PC), Rn and 4-byte literal data
MOVA #imm, R0	MOVA @(disp, PC), R0 and 4-byte literal data

8.3 Literal Pool Output

The literal pool is output to one of the following locations:

- After an unconditional branch (after the delay slot instruction following BRA, JMP, RTS, or RTE)
- Where a .POOL directive has been specified by the programmer

The assembler outputs the literal corresponding to an extended instruction to the nearest output location following the extended instruction. The assembler gathers the literals to be output as a literal pool.

CAUTION!

When a label is specified in a delay slot instruction, no literal pool will be output to the location following the delay slot.

8.3.1 Literal Pool Output after Unconditional Branch (BRA, JMP, RTS, RTE)

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
        MOV.L   #H'FFFF0000,R0
        MOV.W   #H'FF00,R1
        MOV.L   #CD1_START,R2
        MOV     #FF,R3
        RTS
        MOV     R0,R10
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
   1 0000F000                    1       .SECTION CD1,CODE,LOCATE=H'0000F000
   2 0000F000                    2  CD1_START
   3 0000F000 5004               3       MOV.L   #H'FFFF0000,R0
   4 0000F002 1103               4       MOV.W   #H'FF00,R1
   5 0000F004 5205               5       MOV.L   #CD1_START,R2
   6 0000F006 6300               6       MOV     #FF,R3
   7 0000F008 000B               7       RTS
   8 0000F00A 6A03               8       MOV     R0,R10
   9                                     **** BEGIN-POOL ****
  10 0000F00C FF00                       DATA FOR SOURCE-LINE 4
  11 0000F00E 0000                       ALIGNMENT CODE
  12 0000F010 FFFF0000                   DATA FOR SOURCE-LINE 3
  13 0000F014 0000F000                   DATA FOR SOURCE-LINE 5
  14                                     **** END-POOL ****
  15                            9       .END
```

8.3.2 Literal Pool Output to the .POOL Location

If literal pool output location after unconditional branches is not available within the valid displacement range (because the program has a small number of unconditional branches), the assembler outputs error message 402. In this case, a .POOL directive must be specified within the valid displacement range.

The valid displacement range is as follows:

- Word-size operation: 0 to 511 bytes
- Long word-size operation: 0 to 1023 bytes

When a literal pool is output to a .POOL location, a branch instruction is also inserted to jump over the literal pool.

An example of literal pool output is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START
        MOV.L   #H'FFFF0000,R0
        MOV.W   #H'FF00,R1
        MOV.L   #CD1_START,R2
        MOV     #FF,R3
        .POOL
        .END
```

Automatic literal pool generation result (source list)

```
 1 0000F000                    1       .SECTION CD1,CODE,LOCATE=H'0000F000
 2 0000F000                    2  CD1_START:
 3 0000F000 5012               3       MOV.L   #H'FFFF0000,R0
 4 0000F002 110E               4       MOV.W   #H'FF00,R1
 5 0000F004 5216               5       MOV.L   #CD1_START,R2
 6 0000F006 6300               6       MOV     #H'FF,R3
 7 0000F008                    7       .POOL
 8                                     **** BEGIN-POOL ****
 9 0000F008 A006                       BRA TO END-POOL
10 0000F00A 0009                       NOP
11 0000F00C FF00                       DATA FOR SOURCE-LINE 4
12 0000F00E 0000                       ALIGNMENT CODE
13 0000F010 FFFF0000                   DATA FOR SOURCE-LINE 3
14 0000F014 0000F000                   DATA FOR SOURCE-LINE 5
15                                    **** END-POOL ****
16                             8       .END
```

8.4 Literal Sharing

When the literals for several extended instructions are gathered into a literal pool, the assembler makes the extended instructions share identical immediate data.

The following operand forms can be identified and shared:

- Symbol
- Constant
- Symbol ± constant

In addition to the above, expressions that are determined to have the same value at assembly processing may be shared.

However, extended instructions having different operation sizes do not share literal data even when they have the same immediate data.

An example of literal data sharing among extended instructions is shown below.

Source program

```
        .SECTION CD1,CODE,LOCATE=H'0000F000
CD1_START:
        MOV.L   #H'FFFF0000,R0
        MOV.W   #H'FF00,R1
        MOV.L   #H'FFFF0000,R2
        MOV     #H'FF,R3
        RTS
        MOV     R0,R10
        .END
```

↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
  1 0000F000                    1         .SECTION CD1,CODE,LOCATE=H'0000F000
  2 0000F000                    2  CD1_START:
  3 0000F000 5004               3         MOV.L   #H'FFFF0000,R0
  4 0000F002 1103               4         MOV.W   #H'FF00,R1
  5 0000F004 5204               5         MOV.L   #H'FFFF0000,R2
  6 0000F006 6300               6         MOV     #H'FF,R3
  7 0000F008 000B               7         RTS
  8 0000F00A 6A03               8         MOV     R0,R10
  9                                  **** BEGIN-POOL ****
 10 0000F00C FF00                        DATA FOR SOURCE-LINE 4
 11 0000F00E 0000                        ALIGNMENT CODE
 12 0000F010 FFFF0000                    DATA FOR SOURCE-LINE 3,5
 13                                 ****  END-POOL  ****
 14                             9         .END
```

8.5 Literal Pool Output Suppression

When a program has too many unconditional branches, the following problems may occur:

- Many small literal pools are output
- Literals are not shared

In these cases, suppress literal pool output as shown below.

```
        ~
<delayed branch instruction>
<delay slot instruction>
.NOPOOL
        ~
```

Example

Source program

```
        ~
CASE1:
        MOV.L   #H'FFFF0000,R0    ----- Extended instruction 1
        RTS
        NOP
        .NOPOOL                   ----- No literal pool is output here
CASE2:
        MOV.L   #H'FFFF0000,R0    ----- Extended instruction 2
        RTS
        NOP
                                  ----- Literal pool is output here
        ~
```

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

Automatic literal pool generation result (source list)

```
        ~
  20 0000F000                    20 CASE1:
  21 0000F000 5001               21      MOV.L   #H'FFFF0000,R0
  22 0000F002 000B               22      RTS
  23 0000F004 0009               23      NOP
  24                             24     .NOPOOL
  25 0000F006                    25 CASE2:
  26 0000F006 5001               26      MOV.L   #H'FFFF0000,R0
  27 0000F008 000B               27      RTS
  28 0000F00A 0009               28      NOP
  29                                **** BEGIN-POOL ****
  30 0000F00B 0000                     ALIGNMENT CODE
  31 0000F00C FFFF0000                 DATA FOR SOURCE-LINE 21,26
  32                                **** END-POOL   ****
        ~
```

8.6 Notes on Automatic Literal Pool Output

1. If an error occurs when an extended instruction is written

   - Extended instructions must not be specified in delay slots (error 151).
   - Extended instructions must not be specified in relative sections having a boundary alignment value of less than 2 (error 152).
   - MOV.L #imm, Rn or MOVA #imm, R0 must not be specified in relative sections having a boundary alignment value of less than 4 (error 152).

2. If an error occurs when a .POOL directive is written

   .POOL directives must not be written after unconditional branches (error 522).

3. If an error occurs when a .NOPOOL directive is written

   .NOPOOL directives are valid only when written after delay slot instructions. If written at other locations, the .NOPOOL directive causes error 521.

4. If the displacement of an executable instruction exceeds the valid range when an extended instruction is expanded

   The assembler generates a literal pool and outputs error message 402 for the instruction having a displacement outside the valid range.

   Solution: Move the literal pool output location (for example, by the .NOPOOL directive), or change the location or addressing mode of the instruction causing the error.

5. If the literal pool output location cannot be found

   If the assembler cannot find a literal pool output location satisfying the following conditions in respect to the extended instruction,

   - Same file
   - Same section
   - Forward direction

   the assembler outputs, at the end of the section which includes the extended instruction, the literal pool and a BRA instruction with a NOP instruction in the delay slot to jump around the literal pool, and outputs warning message 876.

6. If the displacement from the extended instruction exceeds the valid range

   If the displacement of the literal pool from the extended instruction exceeds the valid range, error 402 is generated.

   Solution: Output the literal pool within the valid range (for example, using the .POOL directive.)

HITACHI

User's Guide

HITACHI

HITACHI

Section 1 Executing the Assembler

1.1 Command Line Format

To start the assembler, enter a command line with the following format when the host computer operating system is in the input wait state.

```
> asmsh Δ <input source file> [,<input source file>...][[Δ] <command line options>...]
  (1)          (2)                                                    (3)
```

(1) Assembler start command.
(2) Name of input source file. Multiple source files can be specified at the same time.
(3) Command line options, which specify the assembly method in more detail.

CAUTION!

When multiple source files are specified on the command line, the unit of assembly processing will be the concatenation of the specified files in the specified order.

In this case, the .END directive must appear only in the last file.

Supplement:

The assembler returns the operating system a return code that reports whether or not the assembly processing terminated normally. The return value indicates the level of the errors occurred as follows.

Normal termination	0
Warnings occurred	0
Errors occurred	MS-DOS: 2
	UNIX: 1
Fatal error occurred	MS-DOS: 4
	UNIX: 1

1.2 File Specification Format

Files handled by the assembler are specified in the following format.

```
<file name>.[<file format>]
```

The term "file name" as used in this manual normally refers to both the file name and the file format.

Example:

(File name)
`file.src` ..... A file with the file name file and the file format src.
`prog.obj` ..... A file with the file name prog and the file format obj.

The file format is used as an identifier to distinguish the contents of the file. Thus two files with differing formats are different files even if the file name is the same.

Example:

`file.src`
`file.obj`
} These file names specify different files.

The assembler handles the following types of file.

- Source file
  This is a source program file. If a source program file is specified without the file format, the file format src will be supplied.

- Object file
  This is an output destination file for object modules. If an object file is specified without the file format, the file format obj will be supplied. If an object file is not specified to the assembler, a file with the same name as the source file (the first specified source file) and with the file format obj will be used.

- Listing file
  This is an output destination file for assemble listings. If a listing file is specified without the file format, the extension lis (UNIX) or lst (MS-DOS) will be supplied. If a listing file is not specified to the assembler, a file with the same name as the source file (the first specified source file) and with the file format lis (UNIX) or lst (MS-DOS) will be used.

  Note: The MS-DOS system treats all-file names, command lines, and subcommand lines as uppercase letters.

Section 2 Command Line Options

2.1 Overview of Command Line Options

Command line options are detailed specifications of the assembly processing. Table 2-1 shows an overview of the command line options.

Table 2-1 Command Line Options

Type	Command Line Option	Function
Object module specifications	–OBJECT –NOOBJECT	Control output of an object module.
	–DEBUG –NODEBUG	Control output of debug information.
Assemble listing specifications	–LIST –NOLIST	Control output of an assemble listing.
	–SOURCE –NOSOURCE	Control output of a source program listing.
	–CROSS_REFERENCE –NOCROSS_REFERENCE	Control output of a cross-reference listing.
	–SECTION –NOSECTION	Control output of a section information listing.
	–SHOW –NOSHOW	Control output of the source program listing.
	–LINES	Sets the number of lines in the assemble listing.
	–COLUMNS	Sets the number of columns in the assemble listing.

CAUTION!

When starting the assembler on MS-DOS, enter a slash (/) instead of a hyphen (-) before the command line options.

Supplement:

The assemble listing is a listing to which the results of the assembly processing are output, and consists of a source program listing, a cross-reference listing, and a section information listing.

References: See appendix C, "Assemble Listing Example", for a detailed description of the assemble listing.

2.2 Command Line Option Reference

2.2.1 Object Module Command Line Options

This assembler provides the following command line options concerned with object modules.

OBJECT	NOOBJECT

These command line options control output of an object module.

DEBUG	NODEBUG

These command line options control output of debug information.

OBJECT	NOOBJECT

Object Module Output Control

Syntax

```
OBJECT [= <object output file>]
NOOBJECT
```

The abbreviated forms are indicated by shading.

Description

1. The OBJECT option specifies output of an object module.

   The NOOBJECT option specifies no output of an object module.

2. The object output file specifies the output destination for the object module.

3. When the object output file parameter is omitted, the assembler takes the following actions:

   - If the file format is omitted:
     The file format obj is supplied.

   - If the specification is completely omitted:
     The file format obj is appended to the name of the input source file (the first specified source file).

4. Do not specify the same file for the input source file and the output object file.

OBJECT	NOOBJECT

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
OBJECT	(regardless of any specification)	An object module is output.
NOOBJECT	(regardless of any specification)	An object module is not output.
(no specification)	.OUTPUT OBJ	An object module is output.
	.OUTPUT NOOBJ	An object module is not output.
	(no specification)	An object module is output.

DEBUG	NODEBUG

Debug Information Output Control

Syntax

DEBUG
NODEBUG

The abbreviated forms are indicated by shading.

Description

1. The DEBUG option specifies output of debug information.

   The NODEBUG option specifies no output of debug information.

2. The DEBUG and NODEBUG options are only valid in cases where an object module is being output.

   References: Object module output
   → Programmer's Guide, 4.2.5, "Object Module Assembler Directives", .OUTPUT
   → User's Guide, 2.2.1, "Object Module Command Line Options", OBJECT NOOBJECT

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
DEBUG	(regardless of any specification)	Debug information is output.
NODEBUG	(regardless of any specification)	Debug information is not output.
(no specification)	.OUTPUT DBG	Debug information is output.
	.OUTPUT NODBG	Debug information is not output.
	(no specification)	Debug information is not output.

DEBUG	NODEBUG

Supplement:

Debug information is information required when debugging a program using the simulator/debugger or the emulator, and is part of the object module. Debug information includes information about source statement lines and information about symbols.

2.2.2 Assemble Listing Command Line Options

This assembler provides the following command line options concerned with the assemble listing.

LIST	NOLIST

These command line options control output of an assemble listing.

SOURCE	NOSOURCE

These command line options control output of a source program listing.

CROSS_REFERENCE	NOCROSS_REFERENCE

These command line options control output of a cross-reference listing.

SECTION	NOSECTION

These command line options control output of a section information listing.

SHOW	NOSHOW

These command line options control output of the source program listing.

LINES

This command line option sets the number of lines in the assemble listing.

COLUMNS

This command line option sets the number of columns in the assemble listing.

LIST	NOLIST

Assemble Listing Output Control

Syntax

```
-LIST [ =<listing output file> ]
-NOLIST
```

The abbreviated forms are indicated by shading.

Description

1. The LIST option specifies output of an assemble listing.

   The NOLIST option specifies no output of an assemble listing.

2. The listing output file specifies the output destination file for the assemble listing.

3. When the listing output file parameter is omitted, the assembler takes the following actions:

   - If the file format is omitted:
     The file format lis is supplied.

   - If the specification is completely omitted:
     The file format lis is appended to the name of the input source file (the first specified source file).

4. Do not specify the same file for the input source file and the listing output file.

LIST	NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
LIST	(regardless of any specification)	An assemble listing is output.
NOLIST	(regardless of any specification)	An assemble listing is not output.
(no specification)	.PRINT LIST	An assemble listing is output.
	.PRINT NOLIST	An assemble listing is not output.
	(no specification)	An assemble listing is not output.

SOURCE	NOSOURCE

Source Program Listing Output Control

Syntax

SOURCE
NOSOURCE

The abbreviated forms are indicated by shading.

Description

1. The SOURCE option specifies output of a source program listing to the assemble listing.

   The NOSOURCE option specifies no output of a source program listing to the assemble listing.

2. The SOURCE and NOSOURCE options are only valid in cases where an assembly listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

SOURCE	NOSOURCE

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
SOURCE	(regardless of any specification)	A source program listing is output.
NOSOURCE	(regardless of any specification)	A source program listing is not output.
(no specification)	.PRINT SRC	A source program listing is output.
	.PRINT NOSRC	A source program listing is not output.
	(no specification)	A source program listing is output.

CROSS_REFERENCE	NOCROSS_REFERENCE

Cross-Reference Listing Output Control

Syntax

```
-CROSS_REFERENCE
-NOCROSS_REFERENCE
```

The abbreviated forms are indicated by shading.

Description

1. The CROSS_REFERENCE option specifies output of a cross-reference listing to the assemble listing.

   The NOCROSS_REFERENCE option specifies no output of a cross-reference listing to the assemble listing.

2. The CROSS_REFERENCE and /NOCROSS_REFERENCE options are only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

CROSS_REFERENCE	NOCROSS_REFERENCE

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
CROSS_REFERENCE	(regardless of any specification)	A cross-reference listing is output.
NOCROSS_REFERENCE	(regardless of any specification)	A cross-reference listing is not output.
(no specification)	.PRINT CREF	A cross-reference listing is output.
	.PRINT NOCREF	A cross-reference listing is not output.
	(no specification)	A cross-reference listing is output.

SECTION	NOSECTION

Section Information Listing Output Control

Syntax

```
SECTION
NOSECTION
```

The abbreviated forms are indicated by shading.

Description

1. The SECTION option specifies output of a section information listing to the assemble listing.

   The NOSECTION option specifies no output of a section information listing to the assemble listing.

2. The SECTION and NOSECTION options are only valid in cases where an assemble listing is being output.

   References: Assemble listing output
   → Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
   → User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

SECTION	NOSECTION

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result (when an assemble listing is output)
SECTION	(regardless of any specification)	A section information listing is output.
NOSECTION	(regardless of any specification)	A section information listing is not output.
(no specification)	.PRINT SCT	A section information listing is output.
	.PRINT NOSCT	A section information listing is not output.
	(no specification)	A section information listing is output.

SHOW	NOSHOW

Source Program Listing Output Control

Syntax

```
<UNIX>
-SHOW [= <output type>[,<output type> ...]]
-NOSHOW [= <output type>[,<output type> ...]]

<MS-DOS>
/SHOW [=(<output type>[,<output type> ...])]
/NOSHOW [=(<output type>[,<output type> ...])]
```

When only one output type is specified, the parentheses can be omitted.

```
Output type:  {CONDITIONALS|DEFINITIONS|CALLS|EXPANSIONS|CODE}
```

The abbreviated forms are indicated by shading.

Description

1. The SHOW option specifies output of preprocessor function source statements and object code lines in the source program listing.

   The NOSHOW option suppresses output of specified preprocessor function source statements and object code display lines in the source program listing.

2. The items specified by output types will be output or suppressed depending on the option. When no output type is specified, all items will be output or suppressed.

   -SHOW: Output
   -NOSHOW: Not output (suppress)

SHOW	NOSHOW

3. The following output types can be specified:

Output Type	Object	Description
CONDITIONALS	Failed condition	Condition-failed .AIF statements
DEFINITIONS	Definition	Macro definition parts, .AREPEAT and .AWHILE definition parts, .INCLUDE directive statements .ASSIGNA and .ASSSIGNC directive statements
CALLS	Call	Macro call statements, .AIF and .AENDI directive statements
EXPANSIONS	Expansion	Macro expansion statements .AREPEAT and .AWHILE expansion statements
CODE	Object code lines	The object code lines exceeding the source statement lines

References: Assemble listing output
→ Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST SOURCE NOSOURCE

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
SHOW=<output type>	(regardless of any specification)	The object code is output.
NOSHOW=<output type>	(regardless of any specification)	The object code is not output.
(no specification)	.LIST <output type> (output)	The object code is output.
	.LIST <output type> (suppress)	The object code is not output.
	(no specification)	The object code is output.

LINES

Sets the Number of Lines in the Assemble Listing

Syntax

```
-LINES=<line count>
```

The abbreviated form is indicated by shading.

Description

1. LINES is the command line option that sets the number of lines on a single page of the assemble listing. The range of valid values for the line count is from 20 to 255.

2. The LINES specification is only valid in cases where an assemble listing is being output.

References: Assemble listing output
→ Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
LINES=<line count>	(regardless of any specification)	The number of lines on a page is given by the LINES specification.
(no specification)	.FORM LIN=<line count>	The number of lines on a page is given by the .FORM specification.
	(no specification)	The number of lines on a page is 60 lines.

Sets the Number of Columns in the Assemble Listing

Syntax

```
COLUMNS=<column count>
```

The abbreviated form is indicated by shading.

Description

1. COLUMNS is the command line option that sets the number of columns in a single line of the assemble listing. The range of valid values for the column count is from 79 to 255.

2. The COLUMNS specification is only valid in cases where an assemble listing is being output.

References: Assemble listing output
→ Programmer's Guide, 4.2.6, "Assemble Listing Assembler Directives", .PRINT
→ User's Guide, 2.2.2, "Assemble Listing Command Line Options", LIST NOLIST

Relationship with Assembler Directives

The assembler gives priority to specifications made with command line options.

Command Line Option	Assembler Directive	Result
COLUMNS=<column count>	(regardless of any specification)	The number of columns in a line is given by the COLUMNS specification.
(no specification)	.FORM COL=<column count>	The number of columns in a line is given by the .FORM specification.
	(no specification)	The number of columns in a line is 132 columns.

HITACHI

Appendix

HITACHI

Appendix A Limitations and Notes on Programming

Table A-1 Limitations and Notes on Programming

No.	Item	Limitation	
1	Character types	ASCII characters	
2	Upper/lower-case letter distinction	Symbols (including section names) Object module names	Distinguished
		Reserved words Executable instruction mnemonics Assembler directive mnemonics Operation sizes Integer constant radixes	Not distinguished
3	Line length	Up to 255 bytes	
4	Program length (in lines)	Up to 65,535 lines	
5	Character constants	Up to 4 characters	
6	Symbol length	Up to 32 characters	
7	Number of symbols	Up to 65,535 symbols	
8	Number of import symbols	Up to 65,535 symbols	
9	Number of export symbols	Up to 65,535 symbols	
10	Section size	Up to H'FFFFFFFF bytes	
11	Number of sections	Up to 65,535 sections	
12	Number of macro generation numbers	Up to 100,000 numbers	
13	Number of literals	Up to 100,000 literals	

Appendix B Sample Program

This appendix presents a sample program written for this assembler.

Sample Program Specifications:

Functional Specification

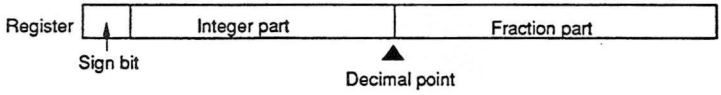
Macros and subroutines for addition, subtraction, multiplication, and division of fixed-point data in the following format:

<parameter 1> OP <parameter 2> → result

OP: +, −, ×, ÷

Note: Operation results are rounded off. Neither underflow nor overflow is checked.

Data Format



The location of the decimal point is set in preprocessor variable POINT as the number of bits from the MSB.

Inputs and Outputs

Inputs: Set parameter 1 in register Parm1.
Set parameter 2 in register Parm2.
For addition and subtraction, parameters 1 and 2 can be specified as macro parameters.

Output: The result is stored in register Parm 1.

Macro and Subroutine Usage

Addition (+): Macro call FIX_ADD [parameter 1], [parameter 2]

Subtraction (−): Macro call FIX_SUB [parameter 1], [parameter 2]

Multiplication (×): Subroutine call FIX_MUL

Division(÷): Subroutine call FIX_DIV

Registers to be Used

Define the following registers with the .REG directive:
Parm1, Parm 2, WORK 1, WORK2, WORK3, WORK4

Supplement

An example of using this sample program is shown in appendix C.

Coding Example:

```
        .MACRO  FIX_ADD Rs=Parm2, Rd=Parm1
        ADD     \Rs,\Rd
        .ENDM

        .MACRO  FIX_SUB Rs=Parm2,Rd=Parm1
        SUB     \Rs,\Rd
        .ENDM

FIX_MUL:
        DIV0S   Parm1,Parm2             ;
        MOVT    WORK1                   ;  Stores the sign of the result in WORK1.
        CMP/PZ  Parm1                   ;  →
        BT      MUL01                   ;    If (Parm1 < 0), Parm1 = –Parm1
        NEG     Parm1,Parm1             ;  →
MUL01   CMP/PZ  Parm2                   ;  →
        BT      MUL02                   ;    If (Parm2 < 0), Parm2 = –Parm2
        NEG     Parm2,Parm2             ;  →
MUL02   MULU    Parm1,Parm2             ;  Parm1 (low) * Parm2 (low)
        SWAP.W  Parm1,Parm1             ;
        STS     MACL,WORK2              ;
        MULU    Parm1,Parm2             ;  Parm1 (high) * Parm2 (low)
        SWAP.W  Parm1,Parm1             ;
        SWAP.W  Parm2,Parm2             ;
        STS     MACL,WORK3              ;
        MULU    Parm1,Parm2             ;  Parm1 (low) * Parm2 (high)
        SWAP.W  Parm1,Parm1             ;
        STS     MACL,WORK4              ;
        MULU    Parm1,Parm2             ;  Parm1 (high) * Parm2( high)
        CLRT                            ;
        STS     MACL,Parm1              ;
        MOV     WORK3,Parm2             ;  →
        SHLR16  WORK3                   ;    |
        SHLL16  Parm2                   ;    |
        ADDC    Parm2,WORK2             ;    |
        ADDC    WORK3,Parm1             ;    |  Sums 16-bit multiplication results.
        MOV     WORK4,Parm2             ;    |
        SHLR16  WORK4                   ;    |
        SHLL16  Parm2                   ;    |
        ADDC    WORK2,Parm2             ;    |
        ADDC    WORK4,Parm1             ;  →
        .AREPEAT \&POINT                ;  →
        SHLL    Parm2                   ;    |  Corrects decimal point location.
        ROTCL   Parm1                   ;    |
        .AENDR                          ;  →
        SHLR    WORK1                   ;  →
        BF      MUL03                   ;;   |  Adds the sign.
        NEG     Parm1,Parm1             ;  →
MUL03   RTS
        NOP
```

(Continued on following page.)

```
FIX_DIV:
        MOV     #0,WORK1                ;  ⟶
        DIV0S   WORK1,Parm1             ;   | If dividend is a negative value,
        SUBC    WORK1,Parm1             ;  ⟶ converts to 1's complement.
        .AREPEAT \&POINT                ;  ⟶
        SHAR    Parm1                   ;   | Corrects decimal point location.
        ROTCR   WORK1                   ;   |
        .AENDR                          ;  ⟶
        DIV0S   Parm2,Parm1             ;
        .AREPEAT 32                     ;  ⟶
        ROTCL   WORK1                   ;   | Parm1:WORK1/Parm2 —> WORK1
        DIV1    Parm2,Parm1             ;   |
        .AENDR                          ;  ⟶
        ROTCL   WORK1                   ;
        MOV     #0,Parm1                ;  ⟶
        ADDC    Parm1,WORK1             ;   | Converts to 2's complement.
        MOV     WORK1,Parm1             ;  ⟶
        RTS
        NOP
```

Appendix C Assemble Listing Output Example

The assemble listing shows the result of the assemble processing. The assemble listing consists of a source program listing, a cross-reference listing, and a section information listing.

This appendix describes the content and output format of the assemble listing using the assembly of the source program shown below as an example. This uses the sample program shown in appendix B to calculate the following:

$$1.5 \times 2.25 + 3 + 5$$

```
POINT   .ASSIGNA 16
Parm1   .REG    (R0)
Parm2   .REG    (R1)
WORK1   .REG    (R2)
WORK2   .REG    (R3)
WORK3   .REG    (R4)
WORK4   .REG    (R5)

        .SECTION SAMPLE,CODE,ALIGN=4
        .INCLUDE "appendix B"

a       .REG    (R8)
b       .REG    (R9)
c       .REG    (R10)
d       .REG    (R11)

start
        STS     PR,@-SP
        MOV.L   #H'00018000,a
        MOV.L   #H'00024000,b
        MOV.L   #H'00030000,c
        MOV.L   #H'00050000,d

        MOV     a,Parm1
        MOV     b,Parm2
        BSR     FIX_MUL
        NOP
        MOV     Parm1,a
        MOV     c,Parm1
        MOV     d,Parm2
        BSR     FIX_DIV
        NOP
        FIX_ADD a
        MOV     Parm1,a
        LDS     @SP+,PR
        RTS
        NOP
        .END
```

C.1 Source Program Listing

The source program listing lists information related to the source statements, including the line number and the corresponding object code.

Figure C-1 shows an example of a source program listing.

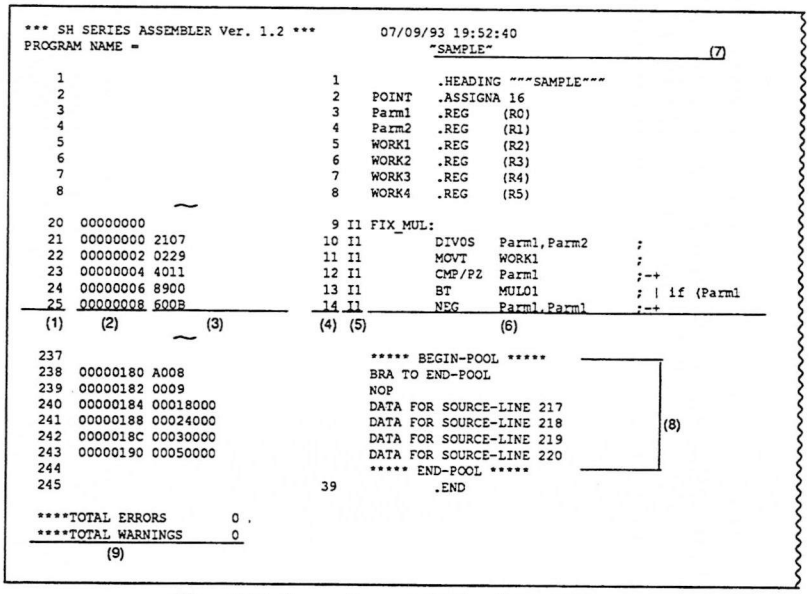


Figure C-1 Source Program Listing Output Example

(1) Line numbers (in decimal)
(2) The value of the location counter (in hexadecimal)
(3) The object code (in hexadecimal). The size of the reserved area in bytes is listed for areas reserved with the .RES, .SRES, .SRESC, and .SRESZ assembler directives.
(4) Source line numbers (in decimal)
(5) Expansion type. Whether the statement is expanded by file inclusion, conditional assembly function, or macro function is listed.
  In: File inclusion (n indicates the nest level).
  C: Satisfied conditional assembly, performed iterated expansion, or satisfied conditional iterated expansion
  M: Macro expansion

(6) The source statements
(7) The header setup with the .HEADING assembler directive.
(8) The literal pool
(9) The total number of errors and warnings. Error messages are listed on the line following the source statement that caused the error.

C.2 Cross-Reference Listing

The cross-reference listing lists information relating to symbols, including the attribute and the value.

Figure C-2 shows an example of a cross-reference listing.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***      07/09/93 19:52:40

*** CROSS REFERENCE LIST

NAME                            SECTION  ATTR VALUE              SEQUENCE

FIX_DIV                         SAMPLE        00000088    94*   229
FIX_MUL                         SAMPLE        00000000    20*   224
MAN03                                   UDEF  00000000    89
MUL01                           SAMPLE        0000000A    24     26*
MUL02                           SAMPLE        00000010    27     29*
Parm1                                   REG                3*    21    23    25
                                                          37    37    39    41
                                                          69    71    73    75
                                                          96    97   102   104
                                                         122   124   126   128
                                                         150   152   154   156
                                                         174   176   178   180
                                                         198   200   202   204
Parm2                                   REG                4*    21    26    28
                                                          44    45    47    49
                                                          70    72    74    76
                                                         144   146   148   150
                                                         168   170   172   174
(1)                             (2)      (3)  (4)          (5)
```

Figure C-2 Cross-Reference Listing Output Example

(1) The symbol name
(2) The name of the section that includes the symbol (first eight characters)
(3) The symbol attribute

EXPT ..... Export symbol
IMPT ..... Import symbol
SCT ..... Section name
REG ..... Symbol defined with the .REG assembler directive
ASGN ..... Symbol defined with the .ASSIGN assembler directive
EQU ..... Symbol defined with the .EQU assembler directive
MDEF ..... Symbol defined two or more times
UDEF ..... Undefined symbol

No symbol attribute (blank)....A symbol other than those listed above

(4) The value of symbol (in hexadecimal)

(5) The list line numbers (in decimal) of the source statements where the symbol is defined or referenced. The line number marked with an asterisk is the line where the symbol is defined.

C.3 Section Information Listing

The section information listing lists information related to the sections in a program, including the section type and section size.

Figure C-3 shows an example of a section information listing.

```
*** SH SERIES ASSEMBLER Ver. 1.2 ***      07/09/93 19:52:40

*** SECTION DATA LIST

SECTION                          ATTRIBUTE    SIZE          START

SAMPLE                           REL-CODE     000000194
  (1)                              (2)          (3)           (4)
```

Figure C-3 Section Information Listing Output Example

(1) The section name

(2) The section type

REL ..... Relative address section
ABS ..... Absolute address section
CODE..... Code section
DATA ..... Data section
COMMON ..... Common section
STACK..... Stack section
DUMMY..... Dummy section

(3) The section size (in hexadecimal, byte units)

(4) The start address of absolute address sections

Appendix D Error Messages

D.1 Error Types

(1) Command Errors

These are errors related to the command line that starts the assembler. These errors can occur, for example, in cases where there are errors in the source file or command line option specifications.

The assembler outputs the error message to standard error output (usually the display). The format of these messages is as follows:

```
<error number><message>
```

Example:

```
10 NO INPUT FILE SPECIFIED
```

(2) Source Program Errors

These are syntax errors in the source program.

The assembler outputs the error message to standard output (usually the display) or the source program listing. (If a source program listing is output during assembly, these messages are not output to standard output.)

The format of these messages is as follows:

```
"<source file name>",line <line number>: ERROR <error number>
"<source file name>",line <line number>: WARNING <error number>
```

Example:

```
"PROG.SRC",line 25: ERROR   300
"PROG.SRC",line 33: WARNING 811
```

The source program error numbers are classified as follows:

100's ..... General source program syntax errors
200's ..... Errors in symbols
300's ..... Errors in operations and/or operands
400's ..... Errors in expressions
500's ..... Errors in assembler directives
600's ..... Errors in file inclusion, conditional assembly, or macro function
800's ..... General source program warnings

(3) Fatal Errors

These are errors related to the assembler operating environment, and can occur, for example, if the available memory is insufficient.

The assembler outputs a message to standard error output. The format of these messages is as follows:

```
FATAL ERROR (<error number>)
```

Example:

```
FATAL ERROR (902)
```

Assembly processing is interrupted when a fatal error occurs.

D.2 Error Message Tables

Table D-1 Command Error Messages

10	Message:	NO INPUT FILE SPECIFIED
	Meaning:	There is no input source file specified.
	Recovery procedure:	Specify an input source file.
20	Message:	CANNOT OPEN FILE <file name>
	Meaning:	The specified file cannot be opened.
	Recovery procedure:	Check and correct the file name and directory.
30	Message:	INVALID COMMAND PARAMETER
	Meaning:	The command line options are not correct.
	Recovery procedure:	Check and correct the command line options.
40	Message:	CANNOT ALLOCATE MEMORY
	Meaning:	All available memory is used up during processing.
	Recovery procedure:	This error only occurs when the amount of available user memory is extremely small. If there is other processing occurring at the same time as assembly, interrupt that processing and restart the assembler. If the error still occurs, check and correct the memory management employed on the host system.
50	Message:	COMPLETED FILE NAME TOO LONG <file name>
	Meaning:	The file name including the directory is too long.
	Recovery procedure:	Shorten the total length of the file name and directory path.
	Supplement:	It is possible that the object module output by the assembler after this error has occurred will not be usable with the simulator/debugger.

Table D-2 Source Program Error Messages

General Source Program Syntax Errors		
100	Error description:	Too complex operation.
	Recovery procedure:	Simplify the expression for the operation.
101	Error description:	Syntax error in executable instruction source statement.
	Recovery procedure:	Check and correct the whole source statement.
102	Error description:	Syntax error in assembler directive source statement.
	Recovery procedure:	Check and correct the whole source statement.
103	Error description:	Program does not end with .END directive.
	Recovery procedure:	Add .END directive.
104	Error description:	The value of location counter exceeded its maximum value.
	Recovery procedure:	Reduce the size of the program.
105	Error description:	Executable instruction or assembler directive that reserves data in stack section.
	Recovery procedure:	Remove the instruction or directive in the stack section.
106	Error description:	Error display terminated due to too many errors.
	Recovery procedure:	Check and correct the whole source statement.
108	Error description:	Illegal continuation line.
	Recovery procedure:	Check and correct continuation line.
109	Error description:	The number of lines being assembled exceeded 65,535 lines.
	Recovery procedure:	Subdivide the program into multiple files.
150	Error description:	Illegal executable instruction placed following delayed branch instruction in memory.
	Recovery procedure:	Change the order of the instruction so that the instruction does not immediately follow a delayed branch instruction.
151	Error description:	Extended instruction placed following a delayed branch instruction in memory.
	Recovery procedure:	Place an executable instruction following the delayed branch instruction.
152	Error description:	Illegal boundary alignment value specified for a section including extended instructions.
	Recovery procedure:	Specify 2 or a larger multiple of 2 as a boundary alignment value.
153	Error description:	Executable or extended instruction placed at an odd address.
	Recovery procedure:	Place executable and extended instructions at even addresses.

Table D-2 Source Program Error Messages (cont)

Symbol Errors		
200	Error description:	Undefined symbol reference.
	Recovery procedure:	Define the symbol.
201	Error description:	Reserved word specified as symbol (or section name).
	Recovery procedure:	Correct the symbol or section name.
202	Error description:	Illegal symbol (or section name).
	Recovery procedure:	Correct the symbol or section name.
Operation and Operand Errors		
300	Error description:	Illegal operation.
	Recovery procedure:	Correct the operation.
301	Error description:	Too many operands of executable instruction, or illegal comment format.
	Recovery procedure:	Check and correct the operands and comment.
304	Error description:	Too few operands.
	Recovery procedure:	Correct the operands.
307	Error description:	Illegal addressing mode in operand.
	Recovery procedure:	Correct the operand.
308	Error description:	Syntax error in operand.
	Recovery procedure:	Correct the operand.
Expression and Operation Errors		
400	Error description:	Character constant is longer than 4 characters.
	Recovery procedure:	Correct the character constant.
402	Error description:	Operand value out of range for this instruction.
	Recovery procedure:	Change the value.
403	Error description:	Attempt to perform multiplication, division, or logic operation on relative value.
	Recovery procedure:	Correct the expression.
407	Error description:	Memory overflow during expression calculation.
	Recovery procedure:	Simplify the expression.
408	Error description:	Attempt to divide by 0.
	Recovery procedure:	Correct the expression.

Table D-2 Source Program Error Messages (cont)

409	Error description:	Register name in expression.
	Recovery procedure:	Correct the expression.
411	Error description:	STARTOF or SIZOF specifies illegal section name.
	Recovery procedure:	Correct the section name.
450	Error description:	Illegal displacement value. (Negative value is specified.)
	Recovery procedure:	Correct the displacement value.
452	Error description:	PC-relative data move instruction specifies illegal address for data area.
	Recovery procedure:	Access a correct address according to the instruction operation size. (4-byte boundary for MOV.L and MOVA, and 2-byte boundary for MOV.W.)
453	Error description:	More than 510 extended instructions exist that have not output literals.
	Recovery procedure:	Output literal pools using .POOL.
Assembler Directive Errors		
500	Error description:	Label not defined in directive that requires label.
	Recovery procedure:	Insert a label.
501	Error description:	Illegal specification of the start address or the value of location counter in section.
	Recovery procedure:	Correct the start address or value location counter.
502	Error description:	Illegal value (forward reference symbol, import symbol, or relative address symbol) specified in operand.
	Recovery procedure:	Correct the operand.
503	Error description:	Symbol declared for export symbol not defined in the file.
	Recovery procedure:	Define the symbol. Alternatively, remove the export symbol declaration.
504	Error description:	Illegal value (forward reference symbol or import symbol) specified in operand.
	Recovery procedure:	Correct the operand.
505	Error description:	Misspelled operand.
	Recovery procedure:	Correct the operand.
506	Error description:	Illegal element specified in operand.
	Recovery procedure:	Correct the operand.

Table D-2 Source Program Error Messages (cont)

508	Error description:	Operand value out of range for this directive.
	Recovery procedure:	Correct the operand.
510	Error description:	Illegal boundary alignment value.
	Recovery procedure:	Correct the boundary alignment value.
512	Error description:	Illegal execution start address.
	Recovery procedure:	Correct the execution start address.
513	Error description:	Illegal register name.
	Recovery procedure:	Correct the register name.
514	Error description:	Symbol declared for export symbol that cannot be exported.
	Recovery procedure:	Remove the declaration for the export symbol.
516	Error description:	Inconsistent directive specification.
	Recovery procedure:	Check and correct all related directives.
517	Error description:	Illegal value (forward reference symbol, an import symbol, or relative-address symbol) specified in operand.
	Recovery procedure:	Correct the operand.
518	Error description:	Symbol declared for import defined in the file.
	Recovery procedure:	Remove the declaration for the import symbol.
521	Error description:	.NOPOOL placed at illegal position.
	Recovery procedure:	Place .NOPOOL following a delayed branch instruction.
522	Error description:	.POOL placed following a delayed branch instruction.
	Recovery procedure:	Place an executable instruction following the delayed branch instruction.
File Inclusion, Conditional Assembly, and Macro Errors		
600	Error description:	Illegal character.
	Recovery procedure:	Correct it.
601	Error description:	Illegal delimiter character.
	Recovery procedure:	Correct it.
602	Error description:	Character string error.
	Recovery procedure:	Correct it.
603	Error description:	Source statement syntax error.
	Recovery procedure:	Reexamine the entire source statement.

Table D-2 Source Program Error Messages (cont)

604	Error description:	Illegal operand specified in a directive.
	Recovery procedure:	No symbol or location counter ($) can be specified as an operand of this directive.
610	Error description:	Macro name reused in macro definition (.MACRO directive).
	Recovery procedure:	Correct the macro name.
611	Error description:	Macro name not specified (.MACRO directive).
	Recovery procedure:	Specify a macro name in the name field of the .MACRO directive.
612	Error description:	Macro name error (.MACRO directive).
	Recovery procedure:	Correct the macro name.
613	Error description:	.MACRO directive appears in macro body (between .MACRO and .ENDM directives), between .AREPEAT and .AENDR directives, or between .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .MACRO directive.
614	Error description:	Identical formal parameters repeated in formal parameter declaration in macro definition (.MACRO directive).
	Recovery procedure:	Correct the formal parameters.
615	Error description:	.END directive appears in macro body (between .MACRO and .ENDM directives).
	Recovery procedure:	Remove the .END directive.
616	Error description:	An .ENDM directive appears without a preceding .MACRO directive, or an .EXITM directive appears outside of a macro body (between .MACRO and .ENDM directives), outside of .AREPEAT and .AENDR directives, or outside of .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .ENDM or .EXITM directive.
618	Error description:	Line with over 255 characters generated by macro expansion.
	Recovery procedure:	Correct the definition or call so that the line is less than or equal to 255 characters.
619	Error description:	Macro parameter name error in macro call, or error in formal parameter in a macro body (between .MACRO and .ENDM directives).
	Recovery procedure:	Correct the formal parameter.
	Supplement:	When there is an error in a formal parameter in a macro body, the error will be detected and flagged during macro expansion.
620	Error description:	Reference to an undefined preprocessor variable.
	Recovery procedure:	Define the preprocessor variable.

Table D-2 Source Program Error Messages (cont)

621	Error description:	.END directive in macro expansion.
	Recovery procedure:	Remove the .END directive.
622	Error description:	Matching parenthesis missing in macro processing exclusion.
	Recovery procedure:	Add the missing macro processing exclusion parenthesis.
623	Error description:	Syntax error in character string manipulation function.
	Recovery procedure:	Correct the character string manipulation function.
624	Error description:	Too many macro parameters for positional specification in macro call.
	Recovery procedure:	Correct the number of macro parameters.
630	Error description:	Syntax error in structured assembly directive operand.
	Recovery procedure:	Reexamine the directive.
631	Error description:	Terminating preprocessor directive does not agree with matching directive.
	Recovery procedure:	Reexamine the preprocessor directives.
640	Error description:	Syntax error in conditional assembly directive operand.
	Recovery procedure:	Reexamine the entire source statement.
641	Error description:	Error in conditional assembly directive relational operator.
	Recovery procedure:	Correct the relational operator.
642	Error description:	.END directive appears between .AREPEAT and .AENDR directives or between .AWHILE and .AENDW directives.
	Recovery procedure:	Remove the .END directive.
643	Error description:	.AENDR or .AENDW directive does not form a proper pair with .AREPEAT or .AWHILE directive.
	Recovery procedure:	Re-examine the preprocessor directives.
644	Error description:	.AENDW or .AENDR directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .AENDW or .AENDR directive.
645	Error description:	Line with over 255 characters generated by .AREPEAT or .AWHILE expansion.
	Recovery procedure:	Correct the .AREPEAT or .AWHILE to generate lines of less than or equal to 255 characters.
650	Error description:	Error in .INCLUDE file name.
	Recovery procedure:	Correct the file name.
651	Error description:	Could not open .INCLUDE file.
	Recovery procedure:	Correct the file name.

Table D-2 Source Program Error Messages (cont)

652	Error description:	File inclusion nesting exceeded 8 levels.
	Recovery procedure:	Limit the nesting to 8 or fewer levels.
653	Error description:	Syntax error in .INCLUDE operand.
	Recovery procedure:	Correct the operand.
660	Error description:	Missing .ENDM directive following .MACRO.
	Recovery procedure:	Insert an .ENDM directive.
662	Error description:	.END directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .END directive.
663	Error description:	.END directive appears in included file.
	Recovery procedure:	Remove the .END directive.
664	Error description:	.END directive appears between .AIF and .AENDI directives.
	Recovery procedure:	Remove the .END directive.
General Source Program Warnings		
800	Error description:	A symbol exceeded 32 characters.
	Recovery procedure:	Correct the symbol.
	Supplement:	The assembler ignores the characters starting at the 33rd character.
801	Error description:	Symbol already defined.
	Recovery procedure:	Remove the symbol redefinition.
	Supplement:	The assembler ignores the second and later definitions.
807	Error description:	Illegal operation size.
	Recovery procedure:	Correct the operation size.
	Supplement:	The assembler ignores the incorrect operation size specification.
808	Error description:	Illegal notation of integer constant.
	Recovery procedure:	Correct the notation.
	Supplement:	The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.
810	Error description:	Too many operands or illegal comment format.
	Recovery procedure:	Correct the operand or the comment.
	Supplement:	The assembler ignores the extra operands.
811	Error description:	Specified label in assembler directive that cannot have a label.
	Recovery procedure:	Remove the label specification.
	Supplement:	The assembler ignores the label.

Table D-2 Source Program Error Messages (cont)

812	Error description:	Section or object module name exceeded 32 characters.
	Recovery procedure:	Correct the section or object module name.
	Supplement:	The assembler ignores the 33rd and later characters.
813	Error description:	A different section type is specified on section restart (reentry), or, a section start address is respecified at the restart of absolute section.
	Recovery procedure:	Do not respecify the section type or start address on section reentry.
	Supplement:	The specification of starting section remains valid.
815	Error description:	Respecification of object module name.
	Recovery procedure:	Specify the object module name once in a program.
	Supplement:	The assembler ignores the second and later object module name specifications.
816	Error description:	Illegal allocation of data or data area.
	Recovery procedure:	Locate the word data or data area on the even address. Locate the long word data or data area on an address of a multiple of 4.
	Supplement:	The assembler corrects the location of the data or data area according to the size of it.
817	Error description:	A boundary alignment value less than 4 specified for a code section.
	Recovery procedure:	The specification is valid, but if an executable instruction or extended instruction is located at an odd address, error 153 occurs.
	Supplement:	Special care must be taken when specifying 1 for code section boundary alignment value.
825	Error description:	Executable instruction or assembler directive that reserves data or data area in dummy section.
	Recovery procedure:	Remove the instruction or directive.
	Supplement:	The assembler ignores the instruction or directive.
832	Error description:	Symbol P already defined before a default section is used.
	Recovery procedure:	Do not define P as a symbol if a default section is used.
	Supplement:	The assembler regards P as the name of the default section, and ignores other definitions of the symbol P.
835	Error description:	Operand value out of range for this instruction.
	Recovery procedure:	Correct the value.
	Supplement:	The assembler generates object code with a value corrected to be within range.

Table D-2 Source Program Error Messages (cont)

836	Error description:	Illegal notation of integer constant.
	Recovery procedure:	Correct the notation.
	Supplement:	The assembler may misinterpret the integer constant, i.e., interpret it as a value not intended by the programmer.
837	Error description:	The length of a source statement exceeded 255 bytes.
	Recovery procedure:	Rewrite the source statement to be within 255 bytes by, for example, rewriting the comment. Alternatively, rewrite the statement as a multi-line statement.
	Supplement:	The assembler ignores byte number 256, and regards the characters starting at byte 257 as the next statement.
850	Error description:	Symbol specified in label field.
	Recovery procedure:	Remove the symbol.
851	Error description:	Macro generation counter exceeded 99999.
	Recovery procedure:	Reduce the number of macro calls.
852	Error description:	Characters appear after the operands.
	Recovery procedure:	Correct the operand(s).
870	Error description:	Illegal displacement value. (Either the displacement value is not an even number when the operation size is word, or the displacement value is not a multiple of 4 when the operation size is long word.)
	Recovery procedure:	Take account of the fact that the assembler corrects the displacement value.
	Supplement:	The assembler generates object code with the displacement corrected according to the operation size. (For a word size operation the assembler discards the low order bit of the displacement to create an even number, and for a long word size operation the assembler discards the two low order bits of the displacement to create a multiple of 4.)
871	Error description:	Executable instruction with PC relative addressing mode operand is located following delayed branch instruction.
	Recovery procedure:	Take account of the fact that the value of PC is changed by a delayed branch instruction.
	Supplement:	The assembler generates object code exactly as specified in the program.

Table D-2 Source Program Error Messages (cont)

874	Error description:	Cannot check data area boundary for PC-relative data move instructions.
	Recovery procedure:	Note carefully the data area boundary at linkage process.
	Supplement:	The assembler only outputs this message when a data move instruction is included in a relative section, or when an import symbol is used to indicate a data area.
875	Error description:	Cannot check displacement size for PC-relative data move instructions.
	Recovery procedure:	Note carefully the distance between data move instructions and data area.
	Supplement:	The assembler only outputs this message for the first illegal instruction in the section.
876	Error description:	The assembler automatically outputs a BRA instruction.
	Recovery procedure:	Specify a literal pool output position using .POOL, or check that the program to which a BRA instruction is added can run normally.
	Supplement:	When a literal pool output location is not available, the assembler automatically outputs literal pool and a BRA instruction to jump over the literal pool.

Table D-3 Fatal Error Messages

901	Error description:	Source file input error.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
902	Error description:	Insufficient memory. (Unable to process the temporary information.)
	Recovery procedure:	Subdivide the program.
903	Error description:	Output error on the list file.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
904	Error description:	Output error on the object file.
	Recovery procedure:	Check the hard disk for adequate free space. Create the required free space by deleting unnecessary files.
905	Error description:	Insufficient memory. (Unable to process the line information.)
	Recovery procedure:	Subdivide the program.
906	Error description:	Insufficient memory. (Unable to process the symbol information.)
	Recovery procedure:	Subdivide the program.
907	Error description:	Insufficient memory. (Unable to process the section information.)
	Recovery procedure:	Subdivide the program.
908	Error description:	The number of sections exceeded 65,535.
	Recovery procedure:	Subdivide the program.
909	Error description:	The number of symbols exceeded 65,535.
	Recovery procedure:	Subdivide the program.
910	Error description:	The number of source program lines exceeded 65,535.
	Recovery procedure:	Subdivide the program.
911	Error description:	The number of import symbols exceeded 65,535.
	Recovery procedure:	Reduce the number of import symbols.
912	Error description:	The number of export symbols exceeded 65,535.
	Recovery procedure:	Reduce the number of export symbols.
950	Error description:	Insufficient memory.
	Recovery procedure:	Separate the source program.
951	Error description:	More than 16 sections exist that have not output literal pools.
	Recovery procedure:	Output literal pools using .POOL before terminating section processing.

Please contact your Hitachi, Ltd., sales representative if a problem cannot be resolved using the indicated recovery procedure, or if an error message that does not appear in the manual is displayed.

Appendix E ASCII Code Table

Table E-1 ASCII Code Table

	Upper 4 Bits							
Lower 4 Bits	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NUL	DLE	SP	0	@	P		p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O		o	DEL

HITACHI

Index

A
absolute address 36
absolute address section 36
  absolute address section declaration 66
absolute address symbol 38
absolute address value 38, 76
absolute path 142, 143
absolute value 38
ADD 51, 54
ADDC 54
address calculation 59
address symbol 16
  absolute address symbol 38
  relative address symbol 38
addressing mode 48
  instruction and operation size combination 52
ADDV 54
.AELSE 155
.AENDI 155
.AENDR 157
.AENDW 159
.AIF 155
.ALIGN 73
AND 51, 55
.AREPEAT 157
arithmetic operation instruction 54
ASCII code table 241
assembler 3, 6
assembler directive 12, 63
assembly-language source program 3, 6
assemble listing 3
  assemble listing assembler directive 119
  assemble listing column count setting 126, 215
  assemble listing command line option 203
  assemble listing line count setting 126, 214
  assemble listing output control 122, 204
  assemble listing output example 223
.ASSIGN 78
.ASSIGNA 150
.ASSIGNC 153
automatic literal generation 185
  extended instruction related to automatic literal generation 186
.AWHILE 159

B
B' 19
backward 41
backward reference 41, 42
BF 56, 58
boundary alignment value
  (adjust the value of the location counter) 21, 73
  section boundary alignment 66
BRA 56, 58
branch instruction 56
  delayed branch instruction 58
BSR 56, 58
BT 56, 58

C
character constant 20
character string 29
  character string data area reservation 100
  character string data area reservation (with length) 102
  character string data area reservation (with zero terminator) 106
  character string data blocks reservation 90
  character string data reservation 88
  character string data reservation (with length) 93
  character string data reservation (with zero terminator) 95
  character string manipulation function 178
CLRMAC 57
CLRT 57
CMP/EQ 51, 54

HITACHI

CMP/GE 54
CMP/GT 54
CMP/HI 54
CMP/HS 54
CMP/PL 54
CMP/PZ 54
CMP/STR 54
code section 31
  code section declaration 66
COLUMNS 215
command line 195
command line option 195, 197
comment 11, 12, 14
  comment for multiple line source statement 15
  common section 33
  common section declaration 66
conditional assembly 146, 155
conditional assembly directive 149
conditional iterated expansion 148, 159
compiler 6, 7
constant 19, 20, 38
constant symbol 38
control character (appended to a character string) 88, 91, 93, 95
CROSS_REFERENCE 208
cross-reference listing 225
  cross-reference listing output control 120, 208

D
D' 19
.DATA 83
data area reservation 97
data section 32
  data section declaration 66
data structure 35
data move instruction 52
.DATAB 85
.DEBUG 117
DEBUG 201
debug information 115, 116
  debug information output control 114
  symbolic debug information output control 117
decrement 49
delay slot instruction 58
delayed branch instruction 58
disp 48, 49, 50
displaced GBR indirect 48
displaced register indirect 48
displacement 49, 50, 62
  correction of displacement 51
divisor of 0 28
DIVOS 54
DIVOU 54
DIV1 54
dummy section 34
  data structure using a dummy section 35
  caution and supplement on dummy section 36, 37

E
editor 6, 7
emulator 7
.END 139
.ENDM 166
error message 227
.EQU 76
executable instruction 47
  executable instruction table 52
executing the assembler 195
.EXITM 161, 169
export 44
  export assembler directive 106
  export symbol 44
  export symbol declaration 44, 107, 111
.EXPORT 107
expression 23
  elements of expression 23
  operator priority 25
  internal processing 28
  notes on expressions 28
extended instruction 53, 186
external 41
external definition 44
external reference 42, 44
external reference value 38
EXTS 54
EXTU 54

F
file format 196
file inclusion 141
file name 196
file specification format 196
.FORM 126
formal parameter 164, 166, 171
  formal parameter default 166
forward 41
forward reference 41, 42

G
GBR 48, 49
GBR indirect with displacement 48
.GLOBAL 111

H
H' 19
header (for source program listing) 128
.HEADING 128

I
#imm 49
immediate 49, 51
import 44
  import assembler directive 106
  import symbol 38, 45
  import symbol declaration 45, 109, 111
.IMPORT 109
.INCLUDE 142
included file 141, 142
inclusion nest 142
increment 49
.INSTR 180
integer constant 19
  integer constant with no radix specification 137
integer data reservation 83
integer data block reservation 85
internal symbol 18
iterated expansion 147, 157

J
JMP 56, 58
JSR 56, 58

K
keyword specification 175

L
label 12, 13
LDC 57
LDS 57
.LEN 178
librarian 6, 7
library file 6, 7
line
  line length 13, 219
  source statements across multiple lines 15
  program length in lines 219
LINES 214
linkage editor 6, 7
.LIST 122
LIST 204
listing file 196
  listing output file 204
load module 6, 7
  S-type format load module 6, 7
location 21, 37
location counter 16, 21
logic operation instruction 55

M
MAC 54
MACH 48, 49
MACL 48, 49
macro 163
  comments in macro 174
  macro body 163, 166, 171
  macro call 175
  macro definition 163, 166
  macro directive 165
  macro function 163
  macro generation number 172
  macro name 163, 166
  macro parameter 164, 175
  macro replacement processing exclusion 173
.MACRO 166
mnemonic 12, 47, 219

assembler directive mnemonic 63
executable instruction
mnemonic 52, 54, 55, 56, 57
MOV 51, 52, 61
MOVA 52, 61
MOVT 52
MULS 54
MULU 54

N
NEG 54
NEGC 54
new page (source program listing) 130
NOCROSS_REFERENCE 208
NODEBUG 201
NOLIST 204
NOOBJECT 199
NOP 57
NOSECTION 210
NOSHOW 212
NOSOURCE 206
NOT 55

O
OBJECT 199
object code 21
object code for delay slot instruction 58
object converter 6, 7
object file 196
object output file 199
object module 3, 6
object module assembler directive 113
object module command line option 198
object module output control 114, 199
object module name 135
operand 11, 12, 14, 47
operation 11, 12, 14, 47
operation (expression) 23
internal processing 28
operation priority 25
operation size 47
instruction and addressing form
combination 52
operator 23
operator association rule 25
operator of reserved word 16
operator priority 25
OR 51, 55
.ORG 71
.OUTPUT 114

P
.PAGE 130
parentheses 24, 25
PC 49, 59, 62
positional specification 175
PR 48, 49
preprocessor variable 145, 150, 153
priority
operation priority 25
operator priority 25
.PRINT 120
.PROGRAM 135
program counter (PC) 16, 49
value of PC 59

Q
Q' 19

R
radix 19
integer constant with no radix
specification 137
.RADIX 137
reassemble 43
.REG 82
register direct 48
register indirect 48
register indirect with index 48
register indirect with post-increment 48
register indirect with pre-decrement 48
register name (reserved word) 16
register name definition 80
relative address 37
relative address section 37
relative address section declaration 66
relative address symbol 38
relative address value 38, 72

relative path 142, 143
relative value 38
.RES 97
reserved word 16, 219
return code 195
Rn 48, 49
ROTCL 56
ROTCR 56
ROTL 56
ROTR 56
RTE 57, 58
RTS 56, 58
R0 48, 49
R15 49

S
sample program 220
.SDATA 88
.SDATAB 90
.SDATAC 93
.SDATAZ 95
.SECTION 66
SECTION 210
section 24, 31, 66
  section attribute (type by usage) 31, 66
  section placement in memory 36, 66
  start and restart of section 67, 68
section information listing 226
  section information listing output control 120, 210
section name 16, 40, 66
section set 24
section set operation 23, 24
separate assembly 43
SETT 57
SHAL 56
SHAR 56
shift instruction 56
SHLL 56
SHLL2 56
SHLL8 56
SHLL16 56
SHLR 56
SHLR2 56
SHLR8 56
SHLR16 56
SHOW 212
simulator/debugger 6, 7
SLEEP 57
SOURCE 206
source file 195, 196
  input source file 195
  multiple source file assembly 195
source program 3, 6
source program listing 224
  source program listing blank line output 132
  source program listing header setting 128
  source program listing new page insertion 130
  source program listing output control 120, 122, 206, 212
source statement 11
  source statement structure 11
  coding of source statement 13
  coding of source statement across multiple lines 15
SP 34, 49
.SPACE 132
SR 48, 49
.SRES 100
.SRESC 102
.SRESZ 104
stack pointer (SP) 34, 49
stack section 34
  caution on stack section 36
  stack section declaration 66
start address for simulation 139
STC 57
STS 57
SUB 54
SUBC 54
.SUBSTR 182
SUBV 54
SWAP 52
symbol (addressing mode) 49, 60, 62
symbol 12
  functions of symbol 16

symbol definition 39
coding of symbol 18
symbol reference 41
symbol debug information 118
symbolic debug information output control 117
system control instruction 57

T
TAS 55
term 23
TRAPA 51, 57, 58
TST 51, 55

U
upper/lower case letter distinction 219

V
value of location counter 21, 39
location counter value setting 71
location counter value correction 73
VBR 48, 49

X
XOR 51, 55
XTRCT 52

SH Series Cross Assembler User's Manual

Publication Date: 1st Edition, August 1993
2nd Edition, March 1994

Published by: Semiconductor and IC Div.
Hitachi, Ltd.

Edited by: Technical Document Center
Hitachi Microcomputer System Ltd.

Copyright © Hitachi, Ltd., 1993. All rights reserved. Printed in Japan.

HITACHI

HITACHI